2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS) | 978-1-6654-7177-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICDCS54860.2022.00111

Implication of Animation on Android Security

Shan Wang†¶, Zhen Ling†*, Yue Zhang‡, Ruizhao Liu†, Joshua Kraunelis‖, Kang Jia†, Bryan Pearson§, Xinwen Fu¶
†Southeast University. Email: shanwangsec@gmail.com,{zhenling,ruizhaoliu,kangjia}@seu.edu.cn
‡Jinan University. Email:zyueinfosec@gmail.com
‖The MITRE Corporation. Email: jkraunelis@mitre.org
§University of Central Florida. Email: bpearson@knights.ucf.edu
¶University of Massachusetts Lowell. Email: xinwen_fu@uml.edu

***Abstract*—We find that seemingly innocuous animations widely used in Android can pose great threats to user security and privacy. Both entrance and exit animations can be exploited. In our draw-and-destroy *overlay* attack, a malicious app periodically draws and destroys transparent UI-intercepting overlays, which can be put over victim apps to intercept user inputs stealthily. Although Android is patched to show alerts if there is an overlay over an app, quickly drawing and destroying malicious overlays can exploit the slow-in animation of the notification alert view and suppress the alert. In our draw-and-destroy *toast* attack, a malicious app periodically creates a new customized toast over a victim app before the previously customized toast disappears. This attack exploits the fade-out animation of the toast so that transition between two successive toasts cannot be observed. The two draw-and-destroy attacks can be building blocks of other attacks. We particularly study the password-stealing attack given its severe consequence, in which the draw-and-destroy *toast* attack displays a fake keyboard over the original keyboard and the draw-and-destroy *overlay* attack places transparent overlays over the fake keyboard to intercept user inputs. Extensive real-world experiments are conducted to validate the feasibility and effectiveness of the attacks. We also discuss defense measures mitigating the attacks. We are the first to discover the security implications of animation on Android security.**

I. Introduction

Animation is a standard element in modern user interface (UI) design [7], [8], [30]. It adds visual cues notifying a user of a view switch and new content, and provides a polished appearance for mobile apps [19]. Immediately view switching with no animation may look disconcerting to users [26]. Animation is also used to defeat attacks such as UI phishing attacks [9]. In such a UI phishing attack, a malicious app creates a fake UI, mimicking and covering the UI of a victim app in a surreptitious way so that a user may fail to notice the fake UI, and type sensitive information such as credentials. With animation, switching from the genuine UI to the fake one may cause flickers and alert the user of phishing attacks.

In this paper we show that the seemingly innocuous animation can be abused and cause security and privacy issues. When Android displays a view, which corresponds to a rectangular area on the screen, it creates a view object for drawing and event handling, and then gradually displays the view with animation. Animation is also used to gradually exit the view. We demonstrate that the *slow-in* or *slow-out* animation of a view can be exploited by two novel Android UI attacks—draw-and-destroy overlay attack and draw-and-destroy toast attack—without raising security alerts.

The slow-in animation can be exploited to launch a novel draw-and-destroy overlay attack suppressing security alerts. In our draw-and-destroy overlay attack, a malicious app periodically performs the following sequence of operations continuously, first drawing an overlay, then waiting for a short period of time (denoted as the attacking window) and finally destroying the overlay. In this way, the malware keeps a sequence of overlays on top of a victim app so as to intercept user inputs. When an overlay is drawn, Android 8.0 and later displays a security alert in the notification drawer as a security mechanism to alert the user and *mitigate known overlay attacks* [1], [6], [33]. However, by carefully controlling the attacking window length, our draw and destroy overlay attack can suppress the security alert. The reason is that the display of the alert uses the slow-in animation. Before the animation could show the alert, the malicious app destroys the overlay and thus stops the animation from showing the alert.

The fade-out animation can be exploited to launch a novel draw-and-destroy toast attack for an extended period of time, defeating Android security mechanism on overlapping toasts. In our draw and destroy toast attack, a malicious app periodically performs the following operations continuously, first creating a customized toast, then waiting for a period of attacking window, and finally creating a new customized toast before Android *automatically* destroys the previous customized toast. In this way, the malware keeps the toast on top of a victim app for an extended period of time. Such a way of abusing animation defeats Android's defense preventing toasts from overlapping each other [18]. The continuous "drawing" and "destroying" of toasts do not cause flickers that may alert a user. The reason is that the disappearance of the toast uses the fade-out animation. Before the toast fades too much and the user may perceive the difference, the malware creates a new toast with the same customized interface.

The two draw-and-destroy attacks exploiting animation can be building blocks of a variety of attacks. For example, a malware may use the draw-and-destroy toast attack to show a fake keyboard while the draw-and-destroy overlay attack can stack transparent overlays over the toasts to intercept user inputs on the fake keyboard. Transparent overlays are legitimate in Android and allow the background to be visible to the user. For example, a transparent/semi-transparent overlay over a map allows a user to see both the map and overlay content.

Contributions: The main contributions of this paper are summarized as follows. *New Insights*: We find that the pervasively used animation in Android causes security and privacy issues. Our draw-and-destroy overlay attack exploits the slow-in animation of notification alerts while the draw-and-destroy toast attack abuses the fade-out animation of toasts.

New Attacks: The discovered novel *draw-and-destroy overlay attack* and *draw-and-destroy toast attack* exploiting animation can be building blocks of a variety of attacks including password stealing, content hiding and payment hijack.

* Corresponding author: Prof. Zhen Ling of Southeast University, China.

We particularly study the password stealing attack given its severe consequence. In our experiments, a password is random and may contain lower case and upper case characters, numbers and special symbols on different sub-keyboards. While the evaluation in this paper focuses on the current most popular Android versions including 8, 9 and 10, the attack works on the newest Android 11 as shown in the anonymous video demo at https://youtu.be/65B2sYHnTiA, which shows the interception of a random password entered on the *Bank of America* app (*BofA*) with a standard toast. Please refer to Section VI-C3 for the description of the video.

Extensive Experiments and User Studies: Extensive real-world experiments are conducted to validate the attacks. Our experiments show that the attacks work against modern Android OSes including the mainstream Android 10 and popular apps such as *Bank of America*, *Skype* and *Facebook*. We conduct human surveys with 30 participants typing passwords on the Bank of America app to evaluate the attack stealthiness of our attacks as presented in Section VI-C3. No participants noticed abnormalities while one person reported slowness using the app. We also use our own test app to collect data such as touch events and touch-event capture rate.

Mitigation: We discuss defense measures, including inter-process communication based and enhanced notification based mechanisms over the *Android Open Source Project (AOSP)* [24] to mitigate the attacks. Experiment results show that the defense measures are effective and the performance overhead is negligible.

Responsible disclosure: We have followed the responsible disclosure policy and reported all our findings to the Google Android Security Team. The Google Android Security Team states that they "have passed the issues on to the feature team for possible remediation."

II. Background

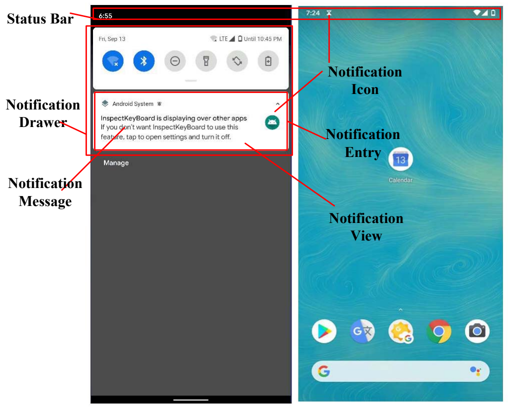
In this section, we introduce the Android *overlay* and *toast* windows, some attacks abusing *overlay* or *toast* and Android built-in defense measures.

A. *Overlay*

In Android, the overlay mechanism provides a capability for an app to draw an overlay window on top of other apps. For example, a music player may use an overlay as a floating widget for users to play/pause music. However, such a mechanism may be abused, and Android has adopted defense measures to mitigate the threats as discussed below.

1) Overlay Attacks: In Draw On Top Attacks [6] against the Android UI, a malicious app may draw an overlay window in the foreground. There are two types of malicious overlays in terms of the goals of the attacker: (1) UI-intercepting overlay: The malware can obtain user inputs such as passwords by using this type of overlay, in which a user interacts with the overlay instead of the underlying victim app. (2) Non-UI-intercepting overlay: This type of overlay can be used to perform a clickjacking attack. When a malware creates an overlay with the attribute of `FLAG_NOT_TOUCHABLE`, touch events pass through the overlay (which does not receive the touch events, unlike the UI-intercepting overlay) to a victim app hidden beneath the malware. The overlay may display misleading contents. When a user acts on the overlay, the user actually interacts with the underlying victim app, e.g., granting administrative privileges via the system Settings app to a malicious app or installing another malicious app [16].

2) Built-in Defenses: Android has the following defense measures to mitigate overlay attacks above: (i) Apps that create foreground overlay windows must request the `SYSTEM_ALERT_WINDOW` permission. (ii) Android 8.0 and later displays a notification message as an alert in a notification entry in the notification drawer when an overlay is present in the foreground [37] as shown in Fig. 1. In Android, a view corresponds to a rectangular area on the screen, and is a basic class for constructing the user interface, in charge of drawing and event handling [27]. The notification entry contains a notification view, which contains the notification message and an icon. The alert can be viewed any time by swiping down on the Android status bar at the top of the screen. The icon can also be viewed at the status bar if there is space. For example, Android 10 of *Google Pixel 2* can show 4 icons at the status bar. To manually remove an unwanted overlay, a user can press on the alert to open the system Settings app, which can prohibit an app from displaying overlays on top of other apps. (iii) Android 8.0 and later prevent any overlay from covering the system Settings app whenever the sytem Settings app is used to grant permissions. The app installer cannot be covered by overlays either.



InspectKeyBoard is displaying over other apps
If you don't want InspectKeyBoard to use this feature, tap to open settings and turn it off.

Fig. 1: The built-in notification defense

B. *Toast*

A toast is a type of view that provides feedback for users. It automatically disappears after a short period of time. The duration of a toast can be set to 2 or 3.5 seconds depending on the preference of the developer. According to Android developer guides [22], when a toast is in the foreground, the underlying activity remains active and visible, and the interaction is not hindered. A toast does not receive touch events and **does not trigger notification alerts.** Therefore, by default a toast is a type of non-UI-intercepting window.

1) Toast Attack: The toast [3], [6] was abused since the toast can be customized with any content and be positioned on the topmost layer without requiring any privileges or triggering the notification alert. For example, the toast can be customized as a keyboard. Recall that the toast lasts for 2 or 3.5 seconds.

To effectively use toasts in attacks, the attacker may want the toast to stay in the foreground as long as possible. An attacker can create a special type of view (i.e., `TYPE_TOAST`), which can stay in the foreground unless removed by the user [6]. An attacker can create a sequence of multiple overlapped toasts by calling the function `Toast.show()` [3]. One toast may appear before the previous toast disappears.

2) Built-in Defense: Android has the following defense measures to mitigate toast attacks: (i) The `TYPE_TOAST` view has been removed since Android 8.0. (ii) Android does not allow the toast to overlap each other anymore, as documented in the change log of Google titled "Prevent apps to overlay other apps via toast windows" [18]). Instead, "the notification manager shows toasts one at a time" [18]. That is, a system service named the notification manager handles all requests of showing toasts in order and shows toasts one after another. The goal here is to insert gaps between multiple toasts so that the user can notice toast attacks [3]. For example, if a sequence of toasts are customized as a keyboard, the user will notice that the keyboard flickers because of the gaps.

III. Draw-and-Destroy Overlay Attack

In this section, we first introduce the threat model. Next, we study the timing of the slow-in animation of the notification alert view since that the timing is critical for the draw-and-destroy overlay attack, which tires to suppress the notification. We then present the workflow of the attack and analyze the parameters affecting the attack.

A. Threat Model

The only assumption for the draw-and-destroy overlay attack is the malicious app is an *overlay* app, which can create overlays on top of other apps. The malicious overlay app may appear like an innocent app and a victim accidentally installs it on a smartphone. Overlay apps are common as shown by the evaluation in Section VI-C2. For example, Google Maps uses the overlay for navigation.

B. Slow-in Animation of Notification Alert

When an app pops up an overlay in the foreground, Android *System UI* calls `startTopAnimation()` to perform the slide down (slow-in) animation and gradually displays the notification view in the notification drawer. The duration of the animation is set to `ANIMATION_DURATION_STANDARD`, which is 360 ms, to display the notification completely [25]. *Interpolator* refers to an animation modifier that "affects the rate of change in an animation" [21]. By default, the mode of the *interpolator* is set to `FastOutSlowInInterpolator`. In this mode, the appearance of the notification view is slow at the beginning and accelerates later under the control of the Bezier curve illustrated in Fig. 2. It can be observed that the animation shows less than 50% of the notification view in the first 100 ms.

The notification view will not be displayed in the first frame of the animation according to the *refresh rate*. *Refresh rate* is used to control the time interval between two continuous frames of the animation. According to Android developer guides, refresh rate is set to 10 ms by default [20]. That is, it takes at least 10 ms to display the first frame of the animation. For example, the height of the notification view is 72 pixels in one of our testing smartphones, *Google Nexus 6P*. The first frame of the animation can only display 0.17% of the notification view according to Fig. 2, i.e., 72 $\times$ 0.17%=0.1224 pixel. The OS rounds up the result of 0.1224 to 0. Accordingly, the notification view will not be displayed in the first frame of the animation.

Since the slow-in animation is a relatively time-consuming task, there exists a period D, denoted as the *attacking window*,

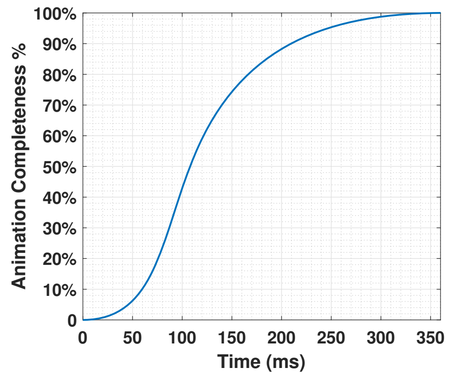


Fig. 2: Time v.s. Percentage of the animation completeness using FastOutSlowInInterpolator

during which the notification alert is not displayed when a malicious app pops up an overlay. The malicious app can continuously draw (add) and destroy (remove) a sequence of overlays to keep the malicious overlays stay in the foreground for an extended period of time with no notification alert displayed. As a result, UI-intercepting overlays can be exploited to stealthily receive user inputs.

C. Attack Workflow

Fig. 3 illustrates the workflow of the *draw-and-destroy overlay attack*.

Step 1: Starting and drawing the first overlay. In our implementation of the draw-and-destroy overlay attack, the main UI thread of the malicious app creates two UI-intercepting overlay objects in advance and also creates a worker thread. Creating the two overlay objects in advance allows accurate control of the timing of the attack since creating objects takes time. The worker thread acts as a timer notifying the main thread through the Android asynchronous handler mechanism to add/display (through function `addView`) and remove (through function `removeView`) the two overlays on the screen .

The first time the main thread is notified, it performs only `addView` to add and display overlay one O_1. The worker thread then enters into the waiting status for a period of attacking window D, which has to be carefully chosen to suppress the notification alert view. `addView` notifies the Android system process *System Server* to add overlay O_1 on the screen. Due to the latency of Interprocess Communication (IPC), it takes a short time period T_{am} for *System Server* to receive the notice from the main thread. It takes T_{as} for *System Server* to create the overlay and add the overlay on the screen. Afterwards, *System Server* notifies another system process *System UI* to draw a notification alert view and calls `startTopAnimation()` to show the view with a slide down animation. It takes T_n for *System Server* to notify the *System UI* of drawing a notification view. The slide down animation requires T_v to prepare the animation and T_a to perform the animation.

Step 2: Destroying current displayed overlay and drawing the other. O_1 is displayed until the worker thread notifies the main thread again. In the second and later rounds, the main thread calls `removeView` to remove O_1. The main thread then calls `addView`, which notifies *System Server* of adding overlay two O_2.

Although `removeView` is called before calling `addView`,

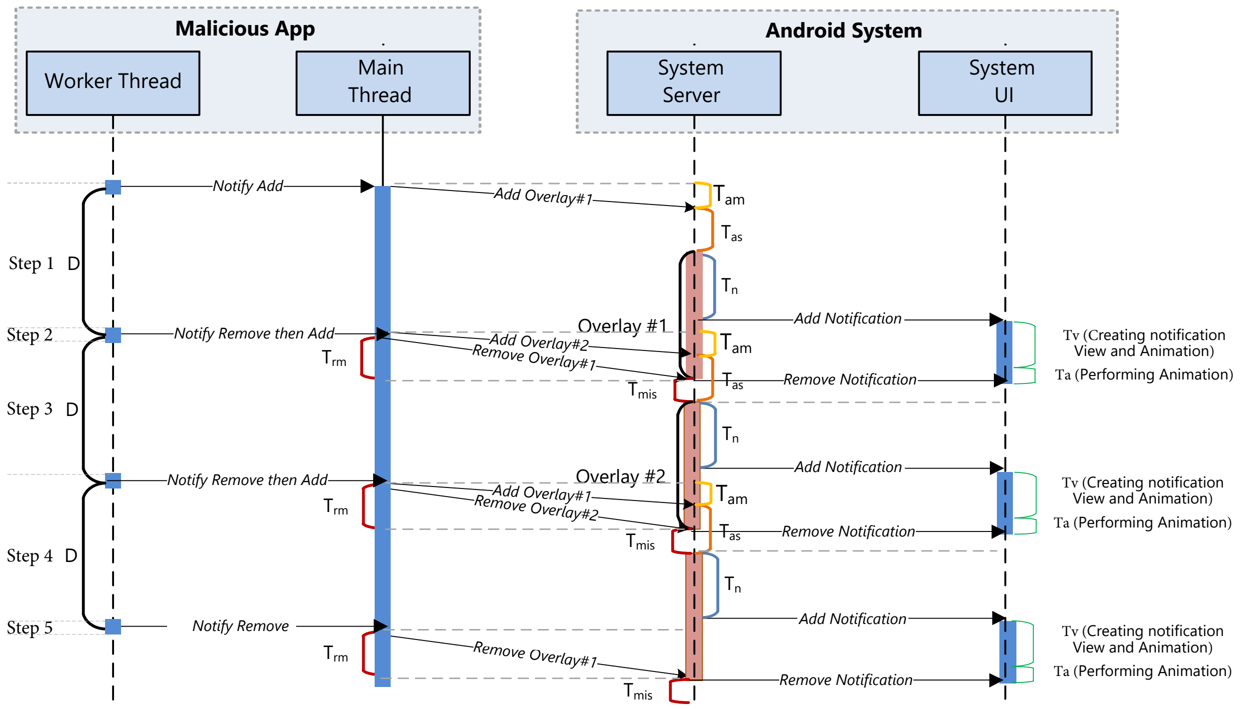


Fig. 3: Entity interaction in the draw and destroy overlay attack

we find that the overlay adding event always reaches *System Server* first. Denote the time period for *System Server* to receive an overlay removing event as T_{rm} and the time period for *System Server* to receive an overlay adding event as T_{am} ($T_{am} < T_{rm}$). When the overlay removing event arrives at *System Server*, *System Server* removes O_1 instantly. After removing O_1, *System Server* checks whether there is still an overlay from the same app in the foreground. If O_2 shows up before O_1 is removed, *System Server* will find there is still an overlay in the foreground and will not notify *System UI* to remove the notification view. If this is the case, *System UI* will continue to play the animation of the notification view.

According to our experiments, the malicious app cannot perform `addView` before `removeView`. `addView` is a blocking function and delays `removeView` from notifying *System Server*. If `addView` is performed before `removeView`, there is a much higher chance that O_2 shows up before O_1 is removed, the notification view animation continues and the attack fails.

Step 3 Waiting for a period of attacking window D. After removing O_1, if *System Server* finds there is no overlay from the same app in the foreground, it notifies *System UI* to remove the notification view. We choose such a small period D that the animation has not shown the notification alert yet when *System UI* is notified to remove the notification view. Therefore, *System UI* stops the slide-down animation and removes the notification view with function `startTopAnimation` in a reverse way.

Step 4 Repeating Steps 2 and 3. Steps 2 and 3 are repeated and the work thread schedules adding and removing the two overlays through the main thread so as to maximize the probability of capturing user inputs on the screen for an extended period of time. Please note: since we add and remove the two overlays O_1 and O_2 in turn, Steps 2 and 3 will be repeated as follows:

Remove O_1 *then add* O_2 $\rightarrow$ *Waiting for* D $\rightarrow$ *Remove* O_2 *then Add* O_1 $\rightarrow$ *Waiting for* D $\rightarrow$ *Remove* O_1 *then add* O_2 $\rightarrow$ *Waiting for* D ...

Step 5 Finishing attack. When the attack is finished, the last displayed overlay is removed.

D. Analysis

Fig. 3 shows that there may exist a gap T_{mis} between the time overlay O_1 is removed and the time overlay O_2 shows up. The malicious app will not be able to capture user touch events since there is no malicious overlay during this gap, that is, mistouches happen. We now analyze this gap, denoted as mistouch duration T_{mis}. $T_{mis} = T_{as} + T_{am} - T_{rm}$, and may vary due to the performance of the overall system. For example, we find that in Android 8 and 9, T_{mis} approaches 0. With $T_{mis} \approx 0$, when the previous overlay is removed, the next overlay can be added immediately. For Android 10 and 11, T_{mis} appears larger and is not negligible.

We now discuss how the attacking window D may affect the chance of "mistouch". Assume that the total attacking period is T, which may vary depending on the ultimate goal of an attacker. For example, if the goal of an attacker is to steal the password of a victim app, the attacker may estimate T based on the typing speed of the user S and the length L of the password, i.e., $T = S \times L$. We assume that the malicious app runs the add/remove operations $n = \lceil \frac{T}{D} \rceil$ times, and discuss a general T in our technical report, which is available on request, while the conclusion is similar. Based on the analysis in Fig.

3, we derive the total mistouch time T_m as follows,

$$T_m = \sum_{i=2}^{n} T_{mis}^{i} + T_{am}^{1} + T_{as}^{1}, \qquad (1)$$

where T_{mis}^{i} is the mistouch time in the ith draw and destroy period. $n > 1$ given that D is small and the attack has to last long enough so as to capture user inputs. Therefore, we can have the expectation of T_m as follows,

$$E(T_m) = (\lceil \frac{T}{D} \rceil - 1)E(T_{mis}) + E(T_{am}) + E(T_{as}). \qquad (2)$$

It can be observed that expected mistouch time $E(T_m)$ decreases as D increases.

Although a large D reduces the mistouch time according to Formula (2), a large D may cause the notification view to be shown on the notification drawer. Therefore, the attacker should carefully choose an upper bound of D. Denote the time used to construct a notification view as T_v. Denote T_a as the time period that the animation plays before the notification view is observable. As show in Fig. 3, to avoid displaying a noticeable notification view in the notification drawer, the attacker has to choose a D less than the time period between the creation of the notification view and the display of the notification view by the animation. That is,

$$D \leq T_n + T_v + T_a. \qquad (3)$$

We derive the maximum D through real-world experiments in Section VI.

IV. Draw-and-Destroy Toast Attack

In this section, we first introduce the threat model. Next, we study of the fade-in and fade-out animation of the toast and the behavior of the animation is critical for the the draw-and-destroy toast attack. We then present the draw-and-destroy toast attack workflow and briefly analyze the impact of the animation on the attack.

A. Threat Model

In this attack, we assume that an attacker only controls a malicious app installed on the victim's smartphone without requiring any sensitive permissions.

B. Animation of Toast View

A toast view is performed with a fade-in animation which plays fast at the beginning and slowly later. *Window Manager Service* performs a fade-in animation to add a toast by calling `startAnimation(.)`. The duration of the animation is set to 500 ms. The animation mode is set to `DecelerateInterpolator`. In this mode, the appearance of the toast view is controlled with an upside-down $y = 1 - (1 - x)^2$ parabola as shown in Fig. 4.

A toast view is removed with a fade-out animation which plays slowly at the beginning and fast later. When it is time for a toast to disappear, the *Window Manager Service* performs a fade-out animation to remove the toast from the screen by calling `startAnimation(.)`. The exit animation lasts for $T_a = 500ms$ and uses the `AccelerateInterpolator` mode. In this mode, the disappearance of the toast view follows a $y = x^2$ parabola as shown in Fig. 4. Due to the slow exit animation, the toast gradually fades out of the screen.

Due to a toast view is displayed fast but removed slowly, a malicious app can continuously creates toasts on the screen so as to display a malicious toast view for an extend period of time. Although Android does not allow overlapping toasts [18], it allows our way of sequentially generating toasts, which "overlap" to a great extent. An attacker can customize the malicious toast with any content (e.g., a keyboard) to deceive users. Please note that a toast view does not trigger notification alerts.

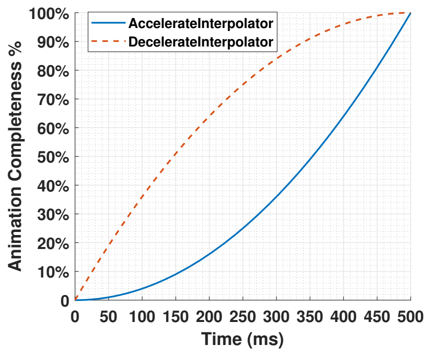


Fig. 4: Time v.s. Percentage of the animation completeness using AccelerateInterpolator and DecelerateInterpolator

C. Attack Workflow

We now elaborate on the detailed workflow of the *draw-and-destroy toast attack* that exploits the animation as shown in Fig. 5. A malicious app continuously creates toasts on the screen and chooses a duration of 2 seconds or 3.5 seconds that Android allows so as to sit on top of a victim app. Since the toast performs a fade-out (slow-out) animation to exit, a new toast can be shown on the screen right before the old one slowly disappears.

Step 1: Creating and displaying a toast. Similar to the draw and destroy *overlay* attack, the malicious app uses a worker thread to control the timing of adding toasts, and the worker thread notifies the main thread via the Android asynchronous handler mechanism. The main thread of the

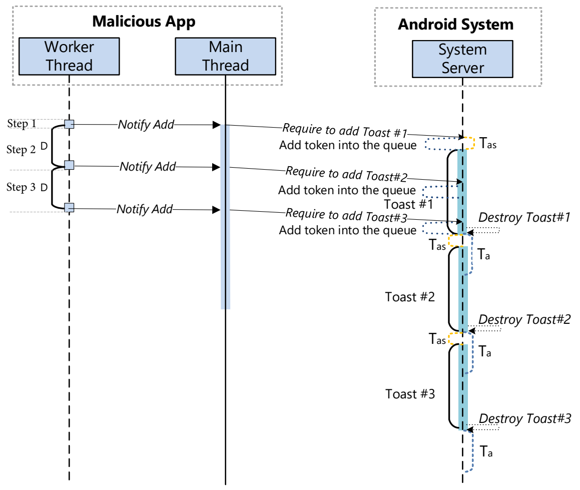


Fig. 5: Entity interaction in the draw and destroy toast attack

malicious app creates a toast, sets the on-screen duration of the toast, and then calls the function `Toast.show()` to notify *System Server*.

The *Notification Manager Service* of *System Server* generates a token and puts the token into a queue via `enqueueToast(.)`. The token uniquely identifies the toast and guarantees that the system does not create a number of overlapping toasts [3].

The *Notification Manager Service* fetches a token from the queue and notifies the *Window Manager Service* of the *System Server* to draw the toast on the screen. Since the *Notification Manager Service* is designed to process one token at a time, the other tokens wait in the queue. In its source code, Android specifies that the number of tokens associated with one app in the queue should be no more than 50. The malicious app can control the time interval D and make sure it generates the required number of toasts for the attack.

When it is time for the toast to disappear, the *Notification Manager Service* invokes the function `removeView(.)`, which notifies the *Window Manager Service* to remove the toast on the screen. Once notified, the *Window Manager Service* performs a **fade-out** animation to remove the toast on the screen by calling `startAnimation(.)`.

Step 2: Waiting for a period of D and creating next toast. The worker thread chooses such a small D that the main thread can create a new toast before the previous one is removed. That is, a new token already exists in the queue before the fade-out animation of the previous toast starts. Therefore, once `removeView(.)` is called, the *System Server* fetches the new token and creates the new toast.

Step 3: Repeating Step 2. The malicious app may repeat Step 2 to keep a toast on top of a victim app for an extended period of time until the attack is completed.

D. Analysis

In Fig. 5, there is a gap T_{as} between two consecutive toasts. T_{as} is the time needed for *System Server* to create a new toast. Despite the existence of T_{as}, users can hardly observe the toast switching due to the **fade-out** animation, as our user study in Section VI shows. To keep the toasts in the foreground for an extended period of time, the attacker shall choose a D and a toast creation strategy so that the toast token queue always has tokens while the number of tokens in the queue is less than 50 at any time during the attack period T. To reduce the number of toast switching within T, the attacker should choose a toast duration of 3.5 s other than 2 s.

V. Password Stealing Attack

The draw-and-destroy *overlay* attack and the draw-and-destroy *toast* attack can be combined to design a sophisticated password stealing attack without alerting users. One challenge of the password stealing attack is to determine when the user enters the password and then the attack is performed. There is related work addressing this challenge, e.g., by means of shared memory side-channel [9] and accessibility service [17].

The detailed workflow of the attack is presented as follows. To launch the attack at the appropriate time, the malicious app determines whether the password input widget of a victim app receives a focus from the user. Upon receiving the focus of the password input widget, the malicious app can deploy both *draw-and-destroy toast attack* and *draw-and-destroy overlay attack* to intercept user inputs. To show a keyboard, the *draw-and-destroy toast attack* is used to implement a fake keyboard covering the real keyboard and switch subkeyboards according to touch events intercepted by the draw-and-destroy overlay attack. The fake keyboard and real keyboard are aligned and appear the same. If the user taps the "*shift*" key on the fake keyboard, the malicious app changes the keyboard view to a new one with the correct subkeyboard layout. If the user taps the "symbol" key such as the key of "*?123*" on the fake keyboard, the malicious app changes the keyboard view to a new one with special symbols. The *draw-and-destroy overlay attack* uses transparent overlays to intercept all user inputs. We implement a callback method on the overlay to capture touch events, which contain the screen coordinates of user touches. Please note: The overlays intercept user inputs, which cannot be passed to the underlying real keyboard. The *draw-and-destroy overlay attack* alone cannot be used as the password stealing attack. Otherwise, the real keyboard underneath the overlays cannot respond to user inputs and switch to subkeyboards.

After obtaining user touch events and the coordinates of the touches, the malicious app can infer the tapped password. The attacker first derives the center coordinate of each key on the real keyboard by performing an offline analysis of the keyboard layout in advance. Then the attacker computes the Euclidean distance between the coordinate of the touched position on the fake keyboard and the center coordinate of each real key. A key is chosen as the typed key if the touched position has the smallest Euclidean distance to the center coordinate of the key.

VI. Experimental Evaluation

In this section, we first present the experiment setup, and then evaluate the draw-and-destroy attacks and the password stealing attack.

A. Experiment Setup

We recruited 30 participants including 5 female and 25 male Android users, and perform real-world user study to evaluate the touch event capture rate of the draw-and-destroy overlay attack, and success rate and stealthiness of the password stealing attacks that utilize the two draw-and-destroy attacks. The age of the participants ranges from 22 to 33 years old and the average age is 25. To evaluate the touch event capture rate, we developed a testing app that adds and removes overlays with an `Activity`. We installed the app on the 30 participants' 30 smartphones as shown in Table I. The brands of these smartphones have the total market share of around 65% [10]. To assess the success rate of our password stealing attacks, we programmed a victim app and a malicious app that performs the two attacks, and also installed them into the smartphones of the participants. We ran the malicious app against 8 different apps such as *Bank of America* and Skype to measure the impact and stealthiness of our attacks. We implemented a crawler and collected 890,855 apps from AndroZoo and developed a static analysis tool based on *aapt* [23] to perform a large-scale analysis of the prevalence of the permission used by our attacks in apps at Android app stores including Google Play.

B. Draw-and-Destroy Overlay Attack

To achieve a high touch event capture rate, we should carefully choose an appropriate value of attacking window D as discussed in Section III so as to defeat the Android built-in notification defense mechanism. According to Equation (2), the expected mistouch time $E(T_m)$ decreases as D increases. However, if D is too large, the notification view may be observed by the user and the attack fails. Therefore, D should

TABLE I: Devices in Evaluation of D

Manufacturer	Model	OS Version
Samsung	s8	8
Samsung	SMG9	9
Google	nexus6p	8
Google	pixel 2xl, pixel 4	9
Google	pixel 2	11
Vivo	v1813A, x21iA, v1816A, v1813BA	9
Vivo	V1986A	10
Oppo	PMEM00	9
Xiaomi	mi5	8
Xiaomi	mix 2s, mi6, mi8	9
Xiaomi	mix3, Redmi, mi8, mi9	10
Xiaomi	mi10	11
Huawei	EML-AL00, mate20, PAR-AL00	9
Huawei	nova3	9.1
Huawei	mate20 x, ELS-AN00, ELE-AL00, OXF-AN00, HLK-AL00	10

have an upper boundary to derive the maximum touch event capture rate.

Upper boundary of D. To determine the upper boundary of D, we try different Ds with the testing app that adds and removes overlays using the 30 smartphones shown in Table I to evaluate whether the notification view could be observed with naked eyes. Fig. 6 shows the five possible outcomes of the notification view with an increasing D. It can be observed that the notification view is a container and shows up first. Other elements in the notification view, including the notification message string and any associated icons, are not displayed until the notification view has been drawn completely. We summarize the outcomes as follows:

Λ_1: The animation does not have an effect yet. No notification view shows up as illustrated in Fig. 6a. This is the most desirable outcome for the attacker.

Λ_2: The animation starts to perform, but is never completed as shown in Fig. 6b. The notification view is partially visible.

Λ_3: The animation is nearly completed. The notification view is fully visible, but no message or icon is displayed in the view as illustrated in Fig. 6c.

Λ_4: The notification view is fully visible, and its associated message is partially displayed in the view as shown in Fig. 6d.

Λ_5: The animation is fully completed. The notification view displays the associated message and icon as illustrated in Fig. 6e. This is the least desirable outcome for the attacker.

Table II shows the upper boundary of D that produces the effect of Λ_1, the best case for the attacker. For brevity, we use the model number and Android version to refer to a smartphone. It can be observed that Android 10 has a greater upper boundary of D compared with Android 8 and Android 9. We explore the source code of Android, and find in Android 10, the time it takes the *System Server* to notify the *System UI* of drawing a notification view (T_n in Fig. 3) is longer than that on Android 8 and Android 9. Android 10 introduces a new service named *Android Notification Assistant (ANA)*, which provides a way for apps to manage notifications. ANA is initialized before the notification is created. Android 10 intentionally introduces a 100 ms (200 ms in Android 11) delay when the *System Server* sends out the notification so as to gain some time for initialization of ANA. As a result, our attack can benefit from the delay and the upper boundary of D is larger for Android 10. According to the analysis in Section III-D, since the performance of different smartphones varies, D is different for distinct phones. To address this issue, the malicious app can collect the phone information before launching the attack so as to select an appropriate upper boundary of D.

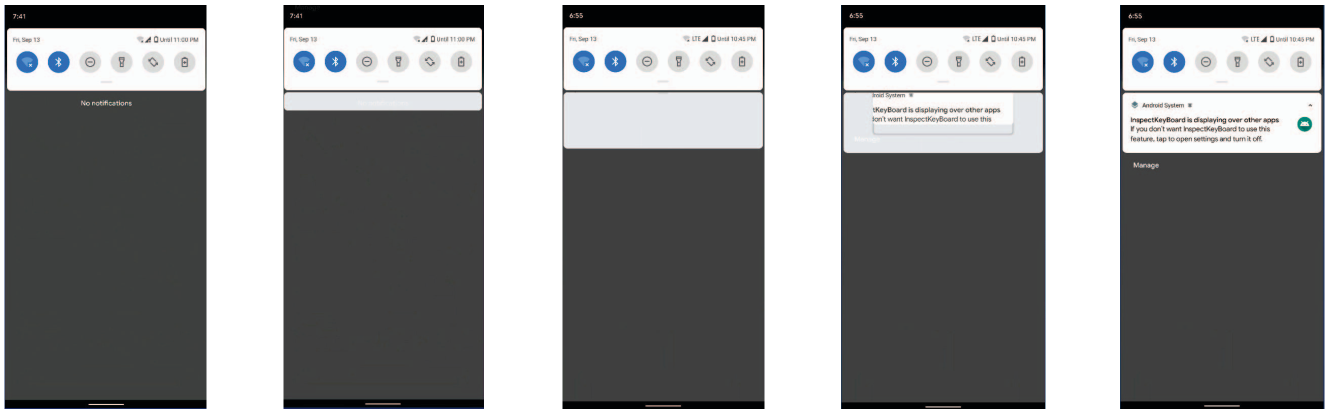
Impact of the load. We have conducted experiments to find how the load of the smartphone can affect the upper boundary of D. We compare three cases in terms of the number of background apps: no background app, three popular apps (i.e., facebook, amazon, and zoom) and five popular apps (i.e., facebook, amazon, zoom, youtube, and twitter). For each case, we perform our attack to evaluate the upper boundary of D. We find that the optimal upper boundaries of D for no app, three apps and five apps in the background are almost the same. Consequently, the influences of the load on the phone is negligible.

TABLE II: Upper boundary of D (ms) on different smartphones

Model	Android Version	Upper boundary of D for Λ_1
s8	8	60
SMG9	9	240
nexus6p	8	150
pixel 2xl	10	225
pixel 4	10	185
pixel 2	11	330
mi5	8	125
mix 2s	9	155
mi8	9	215
mi6	9	215
Redmi	10	395
mi8	10	300
mix3	10	220
mi9	10	210
mi10	11	290
mate20	9	200
EML-AL00	9	365
PAR-AL00	9	130
nova3	9.1	285
mate20 x	10	260
ELS-AN00	10	220
ELE-AL00	10	220
OXF-AN00	10	240
HLK-AL00	10	215
PMEM00	9	135
x21iA	9	85
v1816A	9	95
v1813BA	9	215
v1813A	9	85
V1986A	10	80

Touch event capture rate. We perform a real-world user study to evaluate the touch event capture rate versus D and show the correctness of the theoretical analysis in Section III. The touch event capture rate is the number of touch events captured by the malicious app divided by the total number of touch events. To evaluate the impact of D on the touch event capture rate, D is set to 50 ms, 75 ms, 100 ms, 125 ms, 150 ms, 175 ms, and 200 ms. For each D, each of the 30 participants enters 10 sequences of random strings into an input widget of the testing app on their smartphones. Each random string has 10 characters. Therefore, a total of 100 random characters are entered by each participant. For each D, every participant has a touch event capture rate calculated as the number of captured characters over 100. Please note here we evaluate the touch event capture rate versus D although the notification view/alert may show up with a big D.

Fig. 7 is the box plot showing the impact of D on the touch event capture rate. We label the mean value of the touch event capture rate of the 30 participants for each D. It can be observed that when D increases, the mean value of the touch


No notifications
Android System
InspectKeyBoard is displaying over other apps
If you don't want InspectKeyBoard to use this feature, tap to open settings and turn it off.
Manage

(a) Λ_1 (b) Λ_2 (c) Λ_3 (d) Λ_4 (e) Λ_5

Fig. 6: Possible outcomes of notification view

event capture rate increases. The mean value of the touch event capture rate of 30 participants is around 91% when D reaches around 150 ms. The experiment results match our analysis in Section III.

Fig. 8 gives the impact of Android versions on the touch event capture rate in the draw-and-destroy overlay attack versus D. It shows that the touch event capture rate for Android 10 is lower than that for Android 8 and 9. Android 10 only has a touch event capture rate of around 90% even if D reaches 200 ms. According to our empirical experiment results, T_{rm} in Android 10 is significantly reduced while T_{am} and T_{as} do not change much compared with Android 8 and 9. This phenomenon increases the mistouch duration $T_{mis} = T_{as} + T_{am} - T_{rm}$, and thus the touch event capture rate decreases.

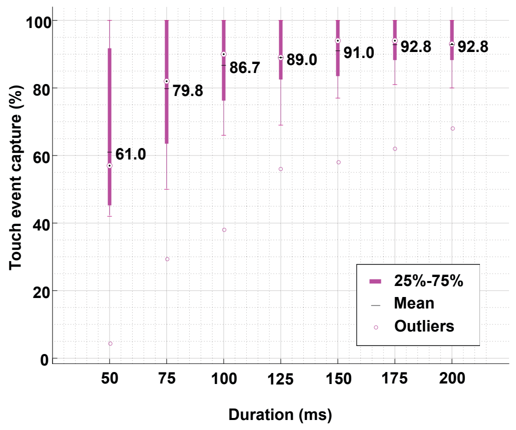


Fig. 7: Touch event capture rate v.s. duration in draw-and-destroy overlay attack

TABLE III: Success rates and errors of the password stealing attack that suppresses the notification view/alert

Password length	4	6	8	10	12
Length errors	10	15	19	23	26
Wrong touched keys	7	8	8	9	9
Capitalization errors	6	7	9	9	12
Success rates	92.3%	90%	88%	86.3%	84.3%

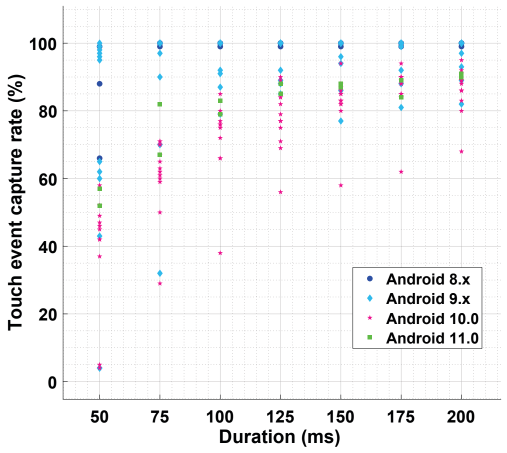


Fig. 8: Android version v.s. touch events capture rate in draw-and-destroy overlay attack

C. Password Stealing Attack

1) Performance: We will evaluate the success rate of the password stealing attack and show it can work against popular real-world apps listed in Table IV.

Success rate. We perform a real-world user study to demonstrate the effectiveness of the password stealing attack using the two draw-and-destroy attacks. In the experiments, a proof of concept malicious app launches the attack to receive the coordinates of user inputs. Toasts are continuously created to mimic a fake software keyboard on the screen and implement the subkeyboard switching functionality according to intercepted keys. We use different upper boundaries of D for different smartphones as shown in Table II to derive the optimal touch event capture rate and avoid being discovered by the users. For the *draw-and-destroy toast attack*, the malicious app uses a fake keyboard with toasts with a duration of 3.5 seconds. We test the password stealing attack against passwords of length 4, 6, 8, 10 and 12. For each length, each of the 30 participants was invited to enter 10 random passwords, which may contain lower and upper case letters, numbers and special symbols on different sub-keyboards. The success rate was the number of successfully obtained passwords divided by the total number of entered passwords.

Table III presents the relationship among three errors, success rates of obtaining the full password and password length. (i) A length error happens when the derived password length is less than the entered password length. A mistouch event of our attack or misspelling by a user may result in such a length error. (ii) A capitalization error is discovered when the length of the derived password is the same as the password required to type, but the case of one or more letters is different. A mistouch event of our attack (i.e., the "shift" key is not captured) or misspelling may result in such an error case. (iii) A wrong touched key error is identified when the derived password length is the same as the entered password length, but one or more letters are different. Misspelling by a user may result in such an error case. The passwords used in these experiments include both uppercase and lowercase letters, and the toasts are used to load different fake subkeyboards if the "shift" key is tapped. The overhead of switching the different keyboards may cause additional delay and result in errors too. Table III shows that our attack can achieve a success rate of 88% with the popular password length of 8 using the draw and destroy attacks. Even if the password length is 12, the success rate is 84.3% for the password stealing attack.

Password stealing attack against real-world apps. We deployed our password stealing attacks against 8 popular apps listed in Table IV and found all of them are subject to our password stealing attacks using the two draw and destroy attacks. Among all the apps, Alipay performs better than others. Alipay is one of the most popular online payment platforms in China. The number of its active users hit 870 million according to a recent report [28], [38]. Alipay disables accessibility events when a user types a password into the password input widget, and our malicious app cannot determine the timing for the attack. Without the accessibility events, the malicious app cannot obtain the object reference of the password input widget and thus cannot fill up the password input widget to hide the attack.

TABLE IV: Apps under testing

App Name	Version	Attacks
Bank of America	8.1.16	✓[a]
Skype	8.45.0.43	✓
Facebook	196.0.0.16.95	✓
Evernote	8.4.1	✓
Snapchat	10.44.3.0	✓
Twitter	7.68.1	✓
Instagram	69.0.0.10.95	✓
Alipay	10.1.65	*[b]

[a] "✓": the tested app can be compromised with no change.
[b] "*": while the app can be compromised, extra efforts are needed.

We are able to defeat the security feature of Alipay as follows. Before a user types a password, the user has to input the username. Alipay does not disable the dispatch of accessibility events sent from the username input widget. This allows us to determine the timing, deploy our attack and derive the object reference of the password input widget: (i) *Identifying the timing to deploy the attack.* When a user interacts with an input widget, a few events indicate the state of the typing progress. When a user starts typing, two events (i.e., `TYPE_VIEW_TEXT_CHANGED` and `TYPE_WINDOW_CONTENT_CHANGED`) are sent by the input widget. When a user finished typing and switches the focus to another widget, only one event (i.e., `TYPE_WINDOW_CONTENT_CHANGED`) was sent by the input widget. The accessibility event `TYPE_WINDOW_CONTENT_CHANGED` sent from the username input widget can be used to indicate the starting time for our attack. (ii) *Obtaining the object reference of the password input widget.* We can obtain the object reference of the password input widget by analyzing the accessibility events sent from the username input widget. Particularly, the username input widget and the password input widget are contained in the same parent view of the Alipay login activity, the object reference of which can be obtained by calling `getParent(.)` of the username input widget. The malicious app can then enumerate each child view of the parent view, and thus identify the object reference of the password input widget. Once we obtain the object reference of the password input widget, we can fill up the password input widget.

2) Legality of Permissions and Methods in Our Attacks: We analyze whether the permission and methods (e.g., *addView(.)* and *removeView(.)*) employed by our attacks are used by the apps in popular app stores so as to see whether our malicious app can have the chance to be hosted in app stores. Our example password stealing attack built on draw and destroy attacks uses *addView(.)*, *removeView(.)*, the overlay permission "`SYSTEM_ALERT_WINDOW`", the *accessibility service* (which is used to detect when the user enters the password. The accessibility service is used as just an example to demonstrate draw and destroy attacks while other approaches can be used to detect when the user enters the password), and a customized toast. We collected 890,855 apps randomly from the AndroZoo [2] database. We build a tool based on *aapt* to statically enumerate the service and permission used in an app. For analysis of the methods, we analyze the 890,855 apps using a tool based on FlowDroid [5]. We find 4,405 apps in 890,855 apps require the permission "`SYSTEM_ALERT_WINDOW`" and register the *accessibility service*. 18,887 apps call both *addView(.)* and *removeView(.)*, and require the permission "`SYSTEM_ALERT_WINDOW`". In addition, there are 15,179 apps using a customized toast. Our experiment indicates that an app store allows apps to use accessibility service, overlays or customized toasts.

3) Stealthiness: We also evaluate the stealthiness of our password stealing attack using the two draw and destroy attacks. Thirty participants were invited to open the *Bank of America* app, type given passwords, and press the *sign in* button. We investigate two scenarios, the smartphone with our malicious app and without. After a participant finished typing passwords, we asked him/her if he/she observed abnormal activities. Only one subject reported that there were lags while he was operating the phone. Except for that, nobody noticed any suspicious thing. We concluded that our attack can be stealthily conducted in the real world scenario.

We create an anonymous video demo at https://youtu.be/65B2sYHnTiA to show the password stealing attack in action on Android 11. In the video, we first show a Google Pixel 2 smartphone uses Android 11 and then grant the needed permissions to our malicious app. The malicious app works in the background to steal a password with lower case and upper case letters, numbers and special symbols. We show the correct captured password (i.e., "tk&%48GH") using a standard toast via our malicious app. In the entire attack process, the victim can hardly identify any abnormality.

VII. Discussion of Defense Measures

In this section, we discuss the use of the interprocess communication (IPC) and other potential defense measures in Android to mitigate the draw and destroy attacks.

A. IPC-based Defense Mechanism

Methodology: In Android, IPC is implemented by the Binder, through which different processes can communicate with each other. For example, an app can call `addView()` and `removeView(.)` methods to notify the System Server of drawing and destroying overlays. Such a call incurs an information-rich Binder transaction, which can be used to determine which method is called as well as the caller, i.e., the app that calls the method. We can change the Binder code (in a minor fashion), collect the Binder transactions of interest and utilize the pattern of the attack to detect and thus terminate them.

We implement a scheme detecting the draw and destroy overlay attack via Android's Binder mechanism using the *Android Open Source Project (AOSP)* [24]. Our detection mechanism works as follows: (i) Collect and forward the collected information including the method caller and timestamp of each Binder transaction of interest to an analyzer; (ii) To detect the draw and destroy overlay attack, the analyzer uses a decision rule, which considers two factors: the number of `addView()` and `removeView()` calls and the duration between a pair of `addView()` and `removeView()` calls.

Experiment results show that the defense measure is effective and the performance overhead is negligible. The details can be found in our technical report, available on request.

B. Enhanced Notification Based Defense Mechanism

In the draw and destroy overlay attack, quickly drawing and destroying malicious overlays can interrupt the display of the notification alert due to the slow-in animation of the alert. To mitigate this issue, we modify the *System Server* to postpone notifying the *System UI* to remove the notification alert. Then the whole alert can be displayed in the notification drawer so that the user can see it and the attack is defeated.

We implement this defense approach using the *Android Open Source Project (AOSP)* [24] of Android 10.0 as follows: (i) When an app invokes `removeView(.)` to destroy an overlay and notify the *System Server*, a delay of $t\ ms$ is added in the *System Server* code before notifying the *System UI* to remove the current notification alert in the notification drawer. (ii) During the delay, if the same app adds a new overlay and notifies the *System Server*, the *System Server* does not notify the *System UI* to remove the alert. Otherwise, the *System Server* notifies *System UI* to remove the alert after the delay. We install our customized AOSP with $t = 690\ ms$ on a Google Pixel 2 and have validated its effectiveness of defeating the draw and destroy overlay attack. To defeat the draw and destroy toast attack, we may change the scheduling algorithm for adding more delay between successive toasts so that the flicker of successively displayed toasts may alert the user.

VIII. Related Work

In this section, we review the most related work on Android UI attacks and defenses.

Android UI attacks. Rydstedt *et al.* [32], [34] demonstrate that mobile browsers are subject to various UI attacks. They design the tap-jacking attacks to steal WPA secret keys and fingerprint the user's geolocation. Our attacks apply to both the browsers and all Android UIs. Felt *et al.* [13] show that UI attacks may go beyond the mobile browsers. For example, in their work, users can be lured to type their sensitive information such as their credentials into a fake mobile login screen controlled by an attacker. Niemietz *et al.* [31] proposed multiple attacks against the Android UI, including the legacy toast attack. Most of those vulnerabilities were already fixed. Chen *et al.* [9] reported that the UI state change can be observed through publicly accessible side channels so that the attackers can pop up a spoofing UI according to the UI state. Bianchi *et al.* [6] analyze multiple scenarios where users can be deceived by a malicious app, such as Draw on top, App switch and Fullscreen. In each scenario, they also list several attack vectors and present a PKI-based framework for UI verification.

Bianchi *et al.* are the first to leverage the overlays to deploy attacks. Since then, overlay-based malware/attacks have been reported in [35], [36], [39]. More recently, Alepis *et al.* [1] show that a transparent overlay activity can cover a victim app, stealing or interfering with user inputs. Yanick *et al.* [16] show that how an app with the overlay mechanism and accessibility service can launch a variety of stealthy and powerful attacks, such as stealing user passwords or installing a malware. To mitigate the overlay abuse above, Android introduces the notification defense and our attacks can defeat such a defense. Simone Aonzo *et al.* reveal that the modern password manager apps and Instant Apps (i.e. apps that can run on the mobile without installation) are vulnerable, and can be abused to design phishing attacks [4]. Our attacks do not relay on the vulnerable password manager apps or Instant Apps.

Android UI secure measures. We now discuss related work on securing UIs. Fernandes *et al.* [14] demonstrate their Trusted Visual I/O Paths (TIVO) prototype. TIVO enables the user to set up a secure image displayed along with the current app name and icon when the user taps and enters data. If the user does not recognize the secure image, or if there is a discrepancy between the secure image and the expected application name and/or icon, then the UI is assumed to have been compromised. This defense may hinder the user experience since a secure image is always displayed in the foreground. Latter, in [15] they proposed a defense against UI attacks in which a notification pops up to tell the user when the background app displays an overlay in the foreground. Android has adopted a similar approach (i.e. the notification view approach) since Android 8 and our draw and destroy overlay attacks work against the defense as discussed in Section III.

Android animation. Animation in UI can improve user experience with both cognitive and affective benefits [12]. Animation can help users better understand what is happening in UI. It can direct attention, provide state-change metaphors and give noticeable feedback on user actions, thus reducing cognitive load and preventing change blindness [11], [29]. Besides, animation makes the visual change on the screen smooth and continuous and can reduce users' uneasiness caused by abrupt visual changes [7]. This makes the user experience more pleasant and comfortable.

IX. Conclusion

We are the first to exploit the seemingly innocuous animation used on mobile devices and design novel UI attacks of severe threats. Particularly, we systematically investigate Android's animation mechanism and present the draw-and-destroy *overlay* attack and the draw-and-destroy *toast* attack, which can be components of a variety of practical attacks such as password stealing. Extensive evaluation and user studies are conducted on popular brands of smartphones such as those from Google and Samsung to validate the discovered attacks. The password stealing attack can achieve a success rates of 88% with the popular password length of 8. To defeat the attacks, We design a detection framework using Android's interprocess communication (IPC) and other mitigation measures such as the enhanced notification based mechanism.

Acknowledgment

This research was supported by National Key R&D Program of China 2018YFB2100300, National Natural Science Foundation of China Grant Nos. 62022024, 61972088, 62072103, 62102084, 62072102, 62072098, and 61972083, by US National Science Foundation (NSF) Awards 1931871, 1915780, and US Department of Energy (DOE) Award DE-EE0009152, by Jiangsu Provincial Natural Science Foundation for Excellent Young Scholars Grant No. BK20190060, Jiangsu Provincial Natural Science Foundation of China under Grant No. BK20190340, Jiangsu Provincial Key Laboratory of Network and Information Security Grant No. BM2003201, Key Laboratory of Computer Network and Information Integration of Ministry of Education of China Grant Nos. 93K-9, and Collaborative Innovation Center of Novel Software Technology and Industrialization. Any opinions, findings, conclusions, and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] E. Alepis and C. Patsakis. Trapped by the ui: The android case. In *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, pages 334–354, Cham, 2017. Springer.

[2] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceeding of the 13th IEEE/ACM Working Conference on Mining Software Repositories (MSR)*, pages 468–471, Austin, TX, USA, 2016. IEEE.

[3] Y. Q. M. T. Analyst). Tapjacking: An untapped threat in android, 2012. [Online]. (Accessed 20 May. 2021).

[4] S. Aonzo, A. Merlo, G. Tavella, and Y. Fratantonio. Phishing attacks on modern android. In *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1788–1801, Toronto, ON, Canada, 2018. ACM.

[5] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[6] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna. What the app is that? deception and countermeasures in the android user interface. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P)*, pages 931–948, San Jose, CA, USA, 2015. IEEE.

[7] A. Chalbi. *Understanding and designing animations in the user interfaces*. PhD thesis, Université lille1, 2018.

[8] B.-W. Chang and D. Ungar. Animation: from cartoons to the user interface. In *Proceedings of the 6th annual ACM symposium on User interface software and technology*, pages 45–55, Atlanta, GA, USA, 1993. ACM.

[9] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into your app without actually seeing it: Ui state inference and novel android attacks. In *Proceedings of the 23rd USENIX Security Symposium (Security)*, pages 1037–1052, San Diego, CA, USA, 2014. IEEE.

[10] Counterpoint. Global smartphone market share: By quarter. https://www.counterpointresearch.com/global-smartphone-share/, 2020. [Online]. (Accessed 20 May. 2021).

[11] C.-E. Dessart, V. G. Motti, and J. Vanderdonckt. Animated transitions between user interface views. In *Proceedings of the International Working Conference on Advanced Visual Interfaces (AVI)*, pages 341–348, Capri Island, Naples, Italy, 2012. ACM.

[12] P. Dragicevic, A. Bezerianos, W. Javed, N. Elmqvist, and J.-D. Fekete. Temporal distortion for animated transitions. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI)*, pages 2009–2018, Vancouver, BC, Canada, 2011. ACM.

[13] A. P. Felt and D. Wagner. *Phishing on mobile devices*. Citeseer, 2011.

[14] E. Fernandes, Q. A. Chen, G. Essl, J. A. Halderman, Z. M. Mao, and A. Prakash. TIVOs: Trusted visual i/o paths for android. Technical Report CSE-TR-586-14, University of Michigan, Ann Arbor, 2014.

[15] E. Fernandes, Q. A. Chen, J. Paupore, G. Essl, J. A. Halderman, Z. M. Mao, and A. Prakash. Android ui deception revisited: Attacks and defenses. In *Proceedings of International Conference on Financial Cryptography and Data Security (FC)*, pages 41–59, Christ Church, Barbados, 2016. Springer.

[16] Y. Fratantonio, C. Qian, S. P. Chung, and W. Lee. Cloak and dagger: from two permissions to complete control of the ui feedback loop. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P)*, pages 1041–1057, San Jose, CA, USA, 2017. IEEE.

[17] Google. Accessibility, 2013. [Online]. (Accessed 20 May. 2021).

[18] Google. Prevent apps to overlay other apps via toast windows, 2016. [Online]. (Accessed 20 May. 2021).

[19] Google. Animations overview, 2019. [Online]. (Accessed 20 May. 2021).

[20] Google. Google Developer API (Fresh Rate), 2019. [Online]. (Accessed 20 May. 2021).

[21] Google. Google Developer API (Interpolators), 2019. [Online]. (Accessed 20 May. 2021).

[22] Google. Toasts overview, 2019. [Online]. (Accessed 20 May. 2021).

[23] Google. Android asset packaging tool. https://developer.android.google.cn/studio/command-line/aapt2?hl=en, 2020. [Online]. (Accessed 20 May. 2021).

[24] Google. Android open source project, 2020. [Online]. (Accessed 20 May. 2021).

[25] Google. Google Developer API (ANIMATION_DURATION_STANDARD), 2020. [Online]. (Accessed 20 May. 2021).

[26] Google. animation, 2021. [Online]. (Accessed 20 May. 2021).

[27] Google. View, 2021. [Online]. (Accessed 20 May. 2021).

[28] A. Group. Alibaba (Website) , 2020. [Online]. (Accessed 20 May. 2021).

[29] V. Head. Designing interface animation. Rosenfeld Media, 2016.

[30] B. Merz, A. N. Tuch, and K. Opwis. Perceived user experience of animated transitions in mobile user interfaces. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems (CHI)*, pages 3152–3158, San Jose, CA, USA, 2016. ACM.

[31] M. Niemietz and J. Schwenk. Ui redressing attacks on android devices. In *Proceedings of Black Hat Abu Dhabi*, pages 1–7, Abu Dhabi, 2012.

[32] S. Rasthofer, I. Asrar, S. Huber, and E. Bodden. An investigation of the android/badaccents malware which exploits a new android tapjacking attack. Technical report, Technische Universität Darmstadt, 2015.

[33] F. Roesner and T. Kohno. Securing embedded user interfaces: Android and beyond. In *Proceedings of the 22nd USENIX Security Symposium (Security)*, pages 97–112, Washington, DC, USA, 2013. USENIX Association.

[34] G. Rydstedt, B. Gourdin, E. Bursztein, and D. Boneh. Framing attacks on smart phones and dumb routers: tap-jacking and geo-localization attacks. In *Proceedings of the 4th USENIX Workshop on Offensive Technologies (WOOT)*, pages 1–8, Washington,D.C., USA, 2010. USENIX Association.

[35] T. Seals. Autorooting, overlay malware are rising android threats, 2016. [Online]. (Accessed 20 May. 2021).

[36] T. Spring. Scourge of android overlay malware on rise, 2016. [Online]. (Accessed 20 May. 2021).

[37] R. Whitwam. Android o feature spotlight: Android tells you if an app is displaying a screen overlay, 2017. [Online]. (Accessed 20 May. 2021).

[38] Wikipad. Alipay(Wikipad), 2019. [Online]. (Accessed 20 May. 2021).

[39] W. Zhou, L. Song, J. Monrad, J. Zeng, and J. Su. The latest android overlay malware spreading via sms phishing in europe, 2016. [Online]. (Accessed 20 May. 2021).