Resource Race Attacks on Android

Yan Cai
State Key Laboratory of Computer Science
Institute of Software
Chinese Academy of Sciences
Beijing, China
ycai.mail@gmail.com

Yutian Tang
Hong Kong Polytechnic University
Hong Kong, China
csytang@comp.polyu.edu.hk

Haicheng Li
State Key Laboratory of Computer Science
Institute of Software
Chinese Academy of Sciences and
University of Chinese Academy of Sciences
Beijing, China
lihc@ios.ac.cn

Le Yu
Hong Kong Polytechnic University
Hong Kong, China
cslyu@comp.polyu.edu.hk

Hao Zhou
Hong Kong Polytechnic University
Hong Kong, China
cshaoz@comp.polyu.edu.hk

Xiapu Luo
Hong Kong Polytechnic University
Hong Kong, China
csxluo@comp.polyu.edu.hk

Liang He
Trust Computing Assurance, Institute of Software
Chinese Academy of Sciences
Beijing, China
heliang@iscas.ac.cn

Purui Su
TCA/SKLCS, Institute of Software, Chinese Academy of Sciences
Cyberspace Security Research Center, Peng Cheng Laboratory
Shenzhen 518000, and
School of Cyber Security, University of Chinese Academy of Sciences
China
purui@iscas.ac.cn

***Abstract*—Smartphones are frequently involved in accessing private user data. Although many studies have been done to prevent malicious apps from leaking private user data, only a few recent works examine how to remove the sensitive information from the data collected by smartphone hardware resources (e.g., camera). Unfortunately, none of them investigates whether a malicious app can obtain such sensitive information when (or right before/after) a legitimate app collects such data (e.g., taking photos). To fill in the gap, in this paper, we model such attacks as the Resource Race Attack (RRAttack) based on races between two apps during their requests to exclusive resources to access sensitive information. RRAttacks have three categories according to when a race on requesting resources occurs: Pre-Use, In-Use, and Post-Use attacks. We further conduct the first systematic study on the feasibility of launching the RRAttacks on two heavily used exclusive Android resources: camera and touchscreen. In details, we perform Proof-of-Concept (PoC) attacks to reveal that, (a) camera is highly vulnerable to both In-Use and Post-Use attacks; and (b) touchscreen is vulnerable to Pre-Use attacks. Particularly, we demonstrate successful RRAttacks on them to steal private information, to cause financial loss, and to steal user passwords from Android 6 to the latest Android Q. Moreover, our analyses on 1,000 apps indicate that most of them are vulnerable to one to three RRAttacks. Finally, we propose a set of defense strategies against RRAttacks for user apps, system apps, and Android system itself.**

***Index Terms*—Resource Race, Android Privacy, Camera, Touchscreen**

I. Introduction

Smartphones have various hardware resources such as cameras and sensors (e.g., GPS), which can be utilized by apps to provide various services to users (e.g., navigation services). As a result, the use of such resources directly involves private user data; and hence, these resources should be well managed.

Android system offers APIs for developers to operate these resources so that developers could determine when and how to utilize a resource. Firstly, this can lead developers to focus more on the functionality but less on the management of these resources, resulting in many bugs (e.g., resource leak which can cause performance degradation and even system crash [1]). Secondly, malicious apps can access to these resources and tamper with private user data, causing security threats [2]–[4]. There have been many works to detect and fix resource leaks [5]–[7] and to detect malicious apps [8]–[12].

Unfortunately, no existing work systematically investigates possible attacks on Android via races on exclusive resources, where an exclusive resource (e.g., a camera) can only be accessed by one app. A race occurs when there are two or more concurrent accesses to the same resource [13], [14]. There have been many such kinds of vulnerabilities [15] like DirtyCow (CVE-2016-5195) which can be exploited to gain root privilege for non-root users. Unfortunately, by causing a race on Android resources, an attacker can also launch many types of attacks as demonstrated in this paper.

We define a **Resource Race** on Android as two concurrent requests to an exclusive resource. For example, according to the API of Camera2 [16], multiple apps can request the same camera at the same time (and the one with a higher priority will be granted to use the camera). That is, there is a race on use of Android cameras. Besides, camera is a preemptive resource. During the use of a camera by an app, a malicious app may preempt the camera without causing any interruption

978-1-7281-5143-4/20/$31.00 © 2020 IEEE

to victim apps. Of course, a preemption on a camera may not bring any security threats but only disturbs user experience; however, considering the environment involved in the camera (i.e., camera view), once there is any private information faced by a camera, security challenges become the first issue to be considered. In other words, considering a camera and its view (e.g., a QR code representing some private information), it may leak user private data when a gained camera is preempted by a malicious app. Besides, a malicious app may also monitor the camera usage and open it immediately after an app releases it. If the camera view remains on the object of interest, the malicious app can launch such attacks by capturing the sensitive information in the camera view.

Considering both environment and resources, we generalize **Resource Race** definition as: two requests to an exclusive resource where the two requests are performed either concurrently or one immediately after another. The latter condition considers two more cases: an app can raise a race on a resource right before another app requests it or right after another app releases it. Then, if an attack can be launched by raising a resource race, we call this attack a **Resource Race Attack** (or **RRAttack** for short). According to the time to raise a race, we propose three basic RRAttack models: Pre-Use attack, In-Use attack, and Post-Use attack. The Pre-Use attack refers to an RRAttack that raises a resource race right before an app requests a resource; the In-Use attack refers to an RRAttack that raises a resource race when the resource is in use; and the Post-Use attack refers to an RRAttack that raises a resource race right after an app releases the resource. We will elaborate more on these attacks in Section II.

In this paper, we conduct the first systematic study on the feasibility of launching RRAttacks on exclusive resources on Android. After analyzing all possible exclusive resources and their management, we successfully identified two resources that are vulnerable to at least one kind of RRAttacks, i.e., cameras and touchscreens. After examining the design of related Android APIs, we conduct proof-of-concept (PoC) attacks to reveal that, (a) camera is highly vulnerable to both In-Use and Post-Use attacks; (b) touchscreen is vulnerable to Pre-Use attacks where a malicious app can create Android windows to catch user inputs. In particular, we demonstrate that RRAttacks can steal private information, cause financial loss, and steal user passwords starting from Android 6 until the latest Android 10 (Q), even though Android Q is designed to provide better privacy protection (e.g., disabling camera usage for background apps [17]).

We further study whether existing apps are vulnerable to RRAttacks and use any protection mechanisms by developing a static analysis tool to inspect 1,000 Android apps. The result shows that, among 337 apps that actually use cameras, 90 apps (26.7%) are vulnerable to Pre-Use attacks on camera and 104 apps (30.9%) are vulnerable to In-Use attacks on camera. Besides, among 224 apps that require passwords (at Login activities), only one is invulnerable to Pre-Use attacks on touchscreen. Besides, more than 81.0% apps that use camera or require password adopt no protection against RRAttacks.

We have reported our RRAttacks to Android Security Team (issue tracker ID: 79652976). Based on our investigation and the experiment results, we find that some of these attacks (e.g., the one explained in Section IV-C) cannot be defended at user app level. **Hence, we claim that, there are several design defects in Android system, which together contribute to RRAttacks, although each separate design does not introduce vulnerability.** Therefore, we propose a set of suggestions to defend against RRAttacks for apps and Android system.

In summary, the major contributions of this paper are:

- We conduct the first systematic study on Resource Race Attacks (RRAttacks) on Android and model three kinds of RRAttacks.
- We reveal that two popular resources on Android are vulnerable to at least one of RRAttacks by (1) investigating the internals of Android system and APIs and (2) conducting successful proof-of-concept attacks on real-world popular apps on several Android platforms, including the latest Android 10 (Q).
- We develop a static analysis tool to check whether existing apps are vulnerable to RRAttacks and employ any protection mechanisms. The results on 1,000 apps show that many apps are vulnerable to RRAttacks and none adopts protection mechanisms. Therefore, we further propose a set of defense suggestions against RRAttacks.

In the rest of this paper, Section II presents the model of RRAttacks in details and discusses possible resources vulnerable to RRAttacks. Sections III and IV show a set of real-world attacks on camera and touchscreen, respectively. Section V presents a study on 1,000 apps against three kinds of RRAttacks. Section VI presents a set of suggestions for defending against RRAttacks. After introducing the related works in Section VII, we conclude the paper in Section VIII.

II. RRAttack Models and Scenarios

A. Why Launching Attacks via Resource Race?

Different from existing attacks on Android (see Section VII), RRAttacks exploit resource races that directly involve highly intensive user interactions and, hence, involve more private information. For example, there will be a higher probability to capture a photo containing private data when a user is taking photos than at other time (e.g., when the camera is idle). In such scenarios, if any race exists and can be exploited, it will cause severe threats. Hence, RRAttacks should also be well studied.

B. Basic Assumptions

In this paper, we focus on races among exclusive resources. A race involves two running apps: one victim app and one malicious app. We assume that both kinds of apps have permissions to access certain Android resources (e.g., camera). We will list permissions required by each RRAttack. Actually, a malicious app can gain permissions relatively easily as users tend to give to an app all the permissions it requests, especially if the app "justifies" its request reasonably. Besides, there are works on how to gain permissions for malicious apps [2], [18].

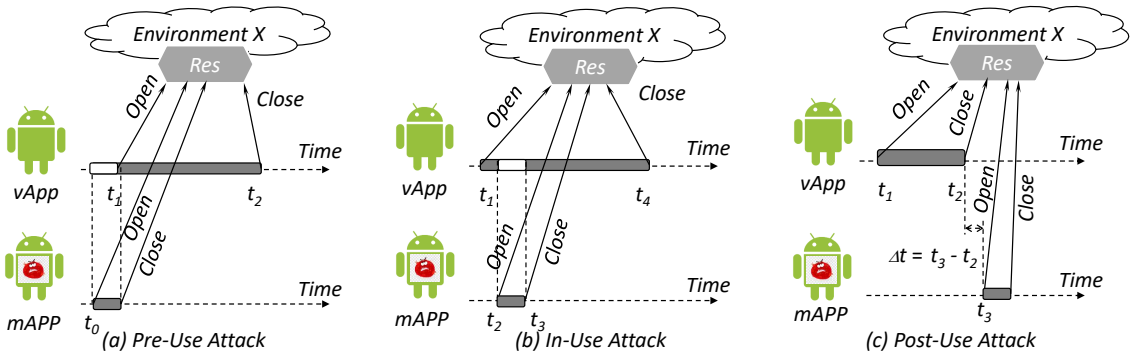

Fig. 1. Three Basic RRAttack Models.

C. Three Attack Models

As explained in Section I, a resource race can be raised in one of three stages. Accordingly, we propose three basic attack models of RRAttack as shown in Figure 1. To simplify our explanation, we use $vApp$ (i.e., $victimApp$) to denote a user app that $opens$ and $closes$ a resource Res to interact with the environment X. We use $mApp$ to denote a malicious app. The three attack models include:

- **Pre-Use Attack.** Suppose that $vApp$ will request a resource Res at time t_1 to interact with environment X and this becomes known to $mApp$. To launch the Pre-Use attack, $mApp$ will firstly request the resource at time t_0, prior to the request by $vApp$. During the period from t_0 to t_1, $mApp$ quickly captures X and then releases Res without disturbing the behavior of $vApp$. If X is clearly captured, $mApp$ can steal any private information in X.
- **In-Use Attack.** This attack involves preemptive resource. That is, when $mApp$ learns that $vApp$ is holding a Res, $mApp$ can preempt Res to quickly capture X and then release Res. This attack requires that, after the preemption on Res by $mApp$, $vApp$ should behave normally (e.g., not crashed). Moreover, the period $\Delta t = t_2 - t_1$ should be short enough.
- **Post-Use Attack.** In this attack, $mApp$ gains the resource Res right after $vApp$ releases it. If the environment X is still available to the resource, $mApp$ can quickly capture the environment.

The three basic models rely on the knowledge of resource usages including when to request, to use, and to release them. A basic and direct requirement is that, a malicious app is able to conduct a quick but short gain on a resource to capture any environment. Further, based on one to three above attacks, attackers can launch sophisticated attacks over a single or multiple resources.

However, various constraints may exist from one smartphone to another for success attacks. We will discuss them and also show several cases in the rest of this paper, including real-world attacks causing financial loss.

D. Resources Vulnerable to RRAttacks

Camera. Cameras are involved in many scenarios in our daily lives (e.g., taking a photo, scanning a QR code). RRAttack on a camera can cause severe effects such as privacy violations. For example, when a user is scanning a ticket, a malicious app can launch an RRAttack to capture the information on the ticket.

According to our investigation and experiment, both In-Use and Post-Use attacks can be easily launched on cameras.

Touchscreen. Touchscreen provides direct interactions between users and apps, such as typing a password. If any RRAttack can be launched during user touches, it can cause severe information leaks, e.g., passwords.

According to our analysis of Android event dispatching and Android Architecture components, Pre-Use attacks can be launched on touchscreens. The corresponding RRAttacks will be presented in Section IV. Fortunately, it may be difficult to launch either In-Use or Post-Use attacks on touchscreens.

Others. Other resources may also be vulnerable to RRAttacks. For example, microphones are usually a kind of exclusive resource. Hence, it is difficult to launch either Pre-Use or In-Use attacks. However, microphones can suffer from Post-Use attacks by immediately opening a microphone to record sounds after it is released by a victim app. In this case, the recorded sound may contain confidential information (e.g., right after a business phone meeting).

III. RRAttacks on Camera

Camera is one of the most important resources on smartphones that can directly capture private data like our faces. Unfortunately, based on our investigation, with only a few carefully designed steps, a malicious app can easily launch RRAttacks on cameras, especially In-Use attacks and Post-Use attacks. We present a technical analysis and then present our implementation and an experiment to validate both In-Use and Post-Use attacks. Note that, Pre-Use attacks on cameras become feasible if one can know in advance when a camera will be opened.

A. Experiment Design

Firstly, to conduct In-Use attacks and Post-Use attacks, we have to precisely monitor camera open and release, respectively. Fortunately, Camera2 provides two exact listener APIs (`onCameraUnavailable()` and `onCameraAvailable()`). These two callbacks are called right after an app opens a camera and right after an app releases a camera, respectively.

The next challenge is how to successfully open a camera after being notified from two listeners. We firstly consider

Fig. 2. Two Tickets and Our Experiment Snapshots.

In-Use attacks on camera. From the source code of the `openCamera()` API, we found that it depends on whether the process (hosting the app) owns the highest user **priority**. Hence, to launch RRAttack, we have to enable our malicious app to have a highest user priority. This is resolved by creating a new foreground activity which automatically obtains the highest priority among user apps. And the activity should not be aware to users (e.g., creating a transparent activity).

After gaining access to an in-use camera, we quickly take several photos, release the camera, and kill the activity. Post-Use attacks can be conducted by following the same steps.

Note that, the most recently released **Android Q** (released in Sep 2019) forbids any background apps to open a camera [17]. However, we only open a camera in a newly created foreground activity, which does not violate this security strategy.

B. Experiment Validation

We have developed a malicious app RRA_{CAM} to monitor cameras based on Camera2 APIs. It is designed as a service with two modes, corresponding to In-Use attack and Post-Use attack. In both modes, after opening the camera, RRA_{CAM} immediately captures 10 photos and then releases it. Of course, RRA_{CAM} is expected to have `CAMERA` permission.

Our experiment is conducted on three Android phones: Huawei Mate 10, LG Nexus 5x, and Google Pixel. They were released in 2013 to 2017, and are installed with Android 7, 6, and Android P and the latest Android Q, respectively. Note, our experiment result on Google Pixel installed with Android P and Q are almost the same; we only report the experimental data below from Android Q.

To firstly validate whether a malicious app can get accurate photos under three attacks, we installed our malicious app RRA_{CAM} on three phones and launched two apps (the default camera app on each phone and a QR code scanner [19]) to scan an airline ticket and a train ticket, respectively. The former contains private text information and the latter contains a QR code (see Figure 2). There are totally 12 groups of experiments. We invited a user to take a photo on the air ticket and to scan the QR code on the train ticket.

Surprisingly, our experiment shows that in all 12 groups of attacks, RRA_{CAM} successfully took clear photos. The only difference is that, on different phones, it took different lengths of time to take the first recognizable photos (see Figure 3).

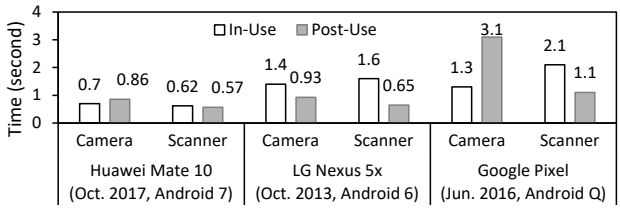

Fig. 3. Time to Take the First Recognizable Photos.

Figure 4 shows photos taken by RRA_{CAM} (only showing the ETKT NO and the QR code in Figure 2 for experiment purpose) and the time cost. We use ✓to indicate photos whether all the texts and QR data can be precisely recognized by an online OCR tool [20] and the QR Code Reader [19], respectively. Although RRA_{CAM} actually took 10 photos, we only show the first five on the first two phones (as at least two of them are recognizable). For Google Pixel, we omit the 2nd to the 6th photos as they are not recognized.

From Figure 4 and Figure 3, we see that, on Huawei Mate 10, it took 0.6 to 0.9 seconds to capture a recognizable photo in both In-Use and Post-Use attacks; on LG Nexus 5x, the cost is from 0.65 to 1.6 seconds; and on Google Pixel, the cost is from 1.1 to 2.1 seconds, except the Post-Use attack in taking photos on air ticket where it cost 3.1 seconds.

In summary, the experiment demonstrates that In-Use and Post-Use attacks on cameras are feasible in practice; they took less than 1.5s for most cases (9/12) (on average, 1.3 seconds).

C. Real-world Attacks

We could identify several scenarios using payments with QR codes in the two most popular social apps in China, i.e., WeChat and QQMobile. Both have the same approach to distribute money as Red Packet (or Red Envelope): one user offers a Red Packet containing an amount of money and this Red Packet is represented by a static QR code. Other accounts can scan this QR code (via the same app) to obtain a random amount of money. The key is that, any user, once owning the QR code, can obtain money until no money is left.

TABLE I
RRATTACK RESULTS ON CAMERA (WECHAT AND QQ).

	WeChat (Static QR)	**WeChat** (Dynamic QR)	**QQ** (Mobile)
In-Use Attack	✓	✓	✓
Post-Use Attack	✓	✗	✓

To verify whether RRAttacks can be combined with above procedure, we set up such a scenario and launched RRAttack when WeChat scans the QR code based on the experiment setup in Section III-B. We invited a third user to scan the QR code captured by our RRA_{CAM} from the server (to simulate an automated attack). The result is as expected, the third user obtained a random amount of money. And WeChat involved in the RRAttack was not aware of either the use of camera by RRA_{CAM} or the "stealing" of money during RRAttacks. We launched the same attack on QQMobile's Red Packet.

We realized in April 2018 that the distribution strategy of Red Packet on WeChat has been enhanced: it regenerates a

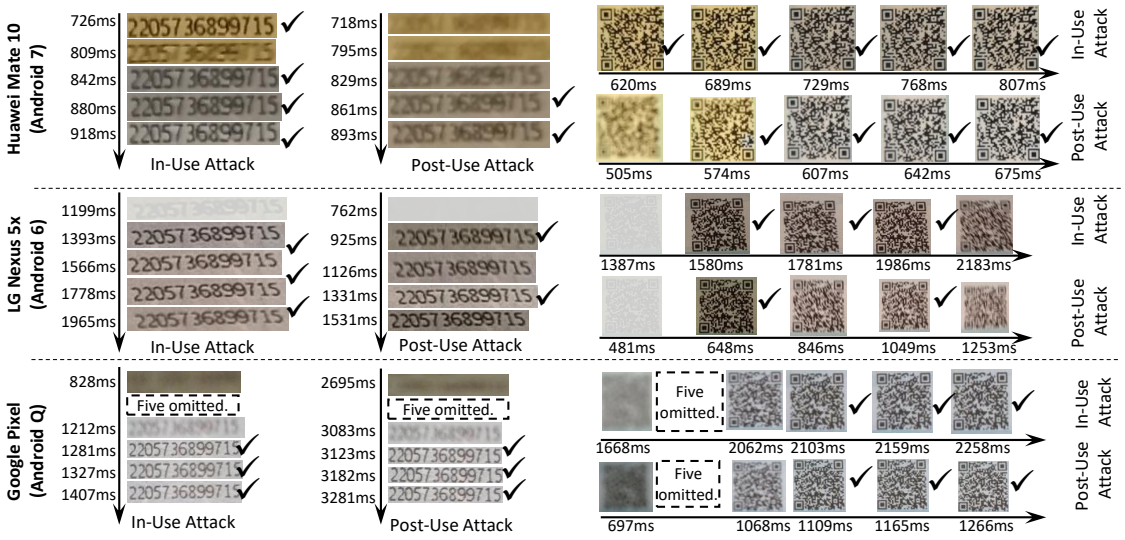

Fig. 4. Results of In-Use and Post-Use Attacks on Three Cameras.

different QR code after a user scans and obtains money from a Red Packet. This makes Post-Use attacks failed as RRA_{CAM} gets an invalid QR code. However, In-Use attack still works and we have successfully verified this (see Table I).

IV. RRAttacks on Touchscreen

A. RRAttack Scenarios on Touchscreens

RRAttacks on touchscreen involve a resource race on user's touches (or touch events) on touchscreen rather than taking touchscreen as a resource. According to the design of Android, one way to launch attacks is to create an Android window [2] to catch user touches and then infer user interaction with the underlying app. Such windows, also known as floating windows or overlays, are different from activities in terms of their responsibilities. Activities are essential components of apps and are responsible for interacting with users [21] while windows are used for placing and presenting views of apps. We first briefly introduce the mechanism for capturing user touches and then describe our attack.

B. Exploitable Design

An app can create one or more windows (i.e., overlays) of customized size and display them above other activities. By default, user touches can be captured by these windows. However, a window can be configured to ignore any user touches by setting the flag `FLAG_NOT_TOUCHABLE`. Thus, any touch inside the window will be delivered to the underlying windows or activities. Moreover, if the flag `FLAG_WATCH_OUTSIDE_TOUCH` is set, the window will receive a `MotionEvent.ACTION_OUTSIDE` event which implies that a user touch event is consumed by another window. The permission required to create the overlay is `SYSTEM_ALERT_WINDOW` and the type of the overlay window is different for different versions of Android (see Table II).

Multiple windows can exist simultaneously, and each of which will be assigned a unique z-order value. Any window setting the flags `FLAG_WATCH_OUTSIDE_TOUCH` and `FLAG_NOT_TOUCHABLE` can receive the same event of the type `MotionEvent.ACTION_OUTSIDE`. The event contains a flag `FLAG_WINDOW_IS_OBSCURED`, which is for security consideration: Android allows an app to know whether there are any other windows shown above it by checking this flag. For example, if there are n windows o_1, o_2 to o_n that have decreasing Z-order values; when a user touches the window o_i, then all windows o_1 to o_i will receive the same event with `FLAG_WINDOW_IS_OBSCURED=0` but the remaining windows o_{i+1} to o_n will receive the same event with `FLAG_WINDOW_IS_OBSCURED=1`.

TABLE II
Types of the Window Required by RRA_{TS1}.

Version < Android O (8.0)	TYPE_SYSTEM_ALERT
Version ≥ Android O (8.0)	TYPE_APPLICATION_OVERLAY

By properly utilizing Android windows, a malicious app may be able to capture user motion actions and then predict user inputs by deliberately arranging a sequence of windows [2]. This vulnerability has been fixed for Android (after 7.1.1); and the flag `FLAG_WINDOW_IS_OBSCURED` of the event `MotionEvent.ACTION_OUTSIDE` will not be set any more [22].

Note, when using overlays (even for normal apps), the *quick-settings dropdown* of the Android phone will list a message to show that an app is laying overlays on other apps. However, Android system does not provide any notifications for users. Therefore, users cannot notice our attack, if they do not check the *quick-settings dropdown* manually.

C. Trap Based User Input Stealing

We illustrate how our attack can steal passwords in Figure 5. When an IME (input method editor) appears, we create a transparent window that covers the IME and set this window to *intercept* touch events (Figure 5(d)), such that the underlying IME cannot receive any touch event. That is, only our transparent window can receive touch events with touched locations

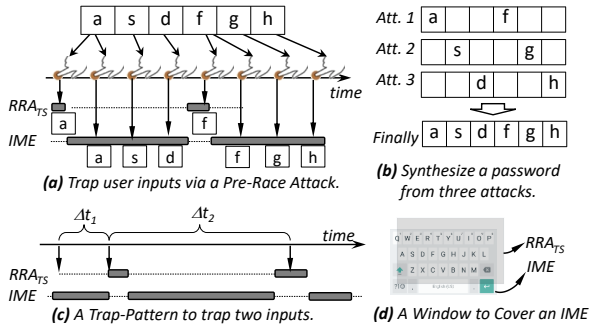

Fig. 5. Trap Based User Inputs Stealing.

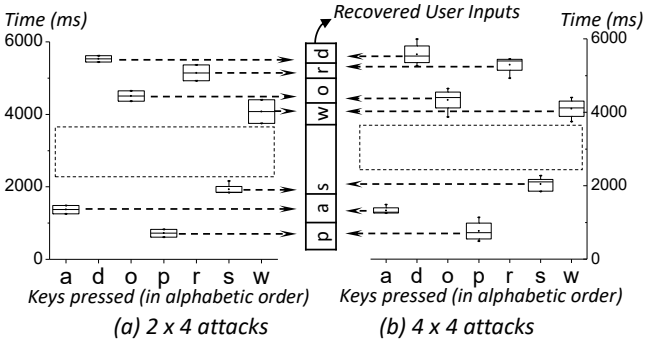

Fig. 6. Result of Trap Based Password Stealing.

which correspond to the key intended to touch by users. We use RRA_{TS1} to denote such a malicious app. Of course, one key step for such attacks is how to know when a user is about to open an IME. This has been investigated and several successful approaches have been proposed based on tracking phones' motion sensors (e.g., accelerometer, gyroscope, and orientation sensors) [23], [24]. As this is not the focus of this paper, we assume that it is known to RRA_{TS1} about when a user opens an IME.

Suppose that the user input consists of six chars: "*asdfgh*". For the first touch (press on key *a*), RRA_{TS1} consumes it; and immediately after that, RRA_{TS1} removes the window. At this time, there is a high probability that users can observe no feedback from the IME (or nothing inputted). Then, the users will touch the same key for the second time. As our window has been removed, the IME can receive the second touch and the key *a* is inputted. In order not to disturb user input too much, RRA_{TS1} will not show the window again until it guesses that the user has pressed several keys. This can be based on statistics over the time elapsed to input one key. Suppose that, after the user presses on key *d*, RRA_{TS1} shows the window for the second time. It can then capture user touch on key *f*. For any later input (of the same session), RRA_{TS1} will not show the window again. Finally, RRA_{TS1} gets two inputs *a* and *f*. In other words, in one attack, RRA_{TS1} is able to capture two keys of a user input as well as the time elapsed in between user touches on the two keys. We call this is a trap-pattern as shown in Figure 5(c).

A trap-pattern requires one or more parameters Δt_i indicating the time to trap the i-th user input (or called i-trap-pattern). In Figure 5(c), the trap-pattern requires two parameters with aim to trap two user inputs in each attack. General, for an i-trap-pattern, it requires at least $l \div i$ attacks to recover a password where l is the length of the password. For the input "*asdfgh*", by carefully design another two trap-patterns, RRA_{TS1} can recover the whole user input (see Figure 5(b)).

We have developed RRA_{TS1} and conducted an experiment to "steal" password. It records the time elapsed from IME appears to the time it traps an input. After running RRA_{TS1}, we invited a user to input "*password*" of 8 chars for 16 input sessions; and RRA_{TS1} is configured to launch each trap-pattern four times. During each session, RRA_{TS1} is configured to trap only two chars. Hence, there are totally 4 trap-patterns. For each trap-pattern, we set Δt_1 to be 0ms, 1000ms, 1500ms, and 2000ms, respectively, and Δt_2 to be 2500ms. The reason for this setting is: we suppose that, after an IME appears, a user usually takes a long time to input the first char (1000ms) but takes a short time to input the subsequent chars (500ms per touch). As our trap-pattern is design to trap a first touch and a fourth touch, there will be four touches; however, once the first touch is trapped, the user has to re-touch the key. Hence, there will be five touches between two trapped touches, corresponding to $5 \times 500ms = 2500ms$.

The result is shown in Figure 6. In each subfigure of Figure 6, the x-axis shows the chars trapped by RRA_{TS1} in alphabetic order, and the y-axis shows the corresponding time elapsed to trap the char in form of a boxplot (as the same trap-pattern is launched multiple times). In detail, Figure 6(a) shows the statistics by applying each trap-pattern twice, i.e., $2 \times 4 = 8$ attacks in the first 8 sessions; and Figure 6(b) shows the same result except where each trap-pattern is launched four times, i.e., $4 \times 4 = 16$ attacks in all 16 sessions.

From the two subfigures, we can conclude that they reflect the same inputs: after an IME appears, the key "*p*" is firstly typed, followed by the keys "*a*", "*s*", "*w*", "*o*", "*r*", and "*d*". From two subfigures, we approximate the user's input to be "*pasword*", although an "*s*" is missing. However, it is obviously reflected by two larger time-span in both subfigures from "*s*" to "*w*" (highlighted by two rectangles). Based on our analysis, this indicates the multiple inputs of the same key. Finally, we can recover the full input "*password*" with a high probability.

This attack can be launched during frequent inputs of the same confidential data (i.e., daily inputs of passwords, PINs, or IDs). For example, many apps (e.g., WeChat Pay and AliPay) only support a six-chars' numeric password (required in each payment). In these scenarios, our attacks will have a high probability to steal the passwords.

Discussion. Our above attack actually utilizes users' *mistakes* during inputting. In practice, it is difficult for a user to *observe* our attacks as it is very common for users to type wrong characters and the inputting error rate is up to 10% [25]. Besides, in our attack, the frequency can also be configured to a low level to eliminate users' suspiciousness. Actually, even if a complete input (e.g., a password) cannot be captured, such attacks can extremely reduce searching space for brute force approaches to guess passwords.

V. Apps Robustness Analysis

To evaluate the robustness of existing apps against three kinds of RRAttacks (Pre-Use attacks on camera and touchscreen, and In-Use attacks on camera), we developed a static analysis tool RRACheck to automatically analyze given apps against three kinds of RRAttacks and applied it on 1,000 popular apps (from `f-droid.org`). We aim to answer the following three research questions:

- **RQ1:** Are existing apps robust against the Pre-Use attacks on camera? What protection mechanisms are used?
- **RQ2:** Are existing apps robust against the In-Use attacks on camera? What protection mechanisms are used?
- **RQ3:** Are apps robust against the Pre-Use attacks on touchscreen? What protection mechanisms are used?

As it is difficult for an app to learn whether any attack may be launched after it releases a camera, we do not study the app robustness against the Post-Use attacks on cameras.

A. Criteria for RQs

A camera can be used in different ways. Usually, its usage pattern is: (1) to create a camera object, (2) to take photos, and (3) to release the camera. As we only focus on Pre-Use and In-Use attacks, we present all possible combinations to create camera objects and to take photos and further determine both Pre-Use and In-Use attacks on camera.

In detail, an app can create a camera object by invoking API calls (e.g., `CameraManager.openCamera()` or `Camera.open()`) under certain scenarios, for example, (1) initializing a camera when an activity is created (e.g., `Activity.onCreate()`), (2) initializing a camera when a surface is created (e.g., `surfaceCreated()`), (3) initializing a camera to response a button event (e.g., `onClick()`) and so forth.

Given the RRAttacks steps and the camera usages, we present two criteria for answering the first two RQs.

Criterion C1 for answering RQ1. When creating camera objects, if an app neither checks the availability of camera nor exception handler to capture whether the focus of the camera is lost unexpectedly, it is vulnerable to Pre-Use attacks on camera.

Assume that an app creates a camera object without checking whether the camera is available. Then, if this camera request is known to a malicious app prior to its creating, the malicious app can easily launch Pre-Use attacks.

Hence, checking availability is part of protection from Pre-Use attacks. Besides, an app can passively listen to camera availability by registering call backs via `CameraManager.registerAvailabilityCallback()`. Additionally, once an expected camera is unavailable, an app can issue a warning to users, allowing users to take further protection.

Criterion C2 for answering RQ2. When taking photos (i.e., using cameras), if an app does not ensure that it is running in foreground, it is vulnerable to In-Use attacks on cameras.

This criteria is straightforward because, under In-Use attack on camera, it is necessary (at least for our validated procedure) to create an activity with a highest priority. This forces the victim app using any camera to be moved in background and to lose the gained camera.

Hence, ensuring running status (being in foreground) is part of protection from In-Use attacks when using cameras. Similarly, additional protection includes issuing a warning to users once an app becomes running in background.

Considering the complexity of Android design, we summarize all concrete implementation to achieve above criteria. Of course, any implementation satisfies above criteria should be regarded as a protection to RRAttacks. On Android, the availability of any camera can be verfied by: (1) checking camera object against `null` right after creating it, (2) capturing exceptions in creating and using cameras (where the two main kinds of exceptions are `RuntimeException` and `CameraAccessException`), and (3) checking whether a device has a camera (`PackageManager.hasSystemFeature()`).

Once the above camera availability checking returns false, an app should notify users. On Android, the notification can be sent to users by either (1) showing a popup window, (2) showing a dialog, or (3) showing a toast window.

Hence, for an app to be invulnerable to RRAttacks for an app, both camera availability and notifications to users should be well designed. We analyze the combinations of above implementations as shown in Table III. In the table, we use "Alert" to denote that at least one kind of three notifications is adopted. We also assign each combination a risk level from 0 to 5, indicating to what extent, the combination suffers from (Pre-Use and In-Use) RRAttacks. A larger risk level indicates that the combination is more likely to suffer from RRAttacks. Totally, there are 19 combinations. For example, for an app implementing "null checking + Exception handling + Alert" is unlikely to suffer from RRAttacks; but an app with no camera availability checking, no exception handling, and of course no notifications to users are most likely to be attacked.

TABLE III
Risk Level for RRAttacks.

#	Programming Pattern	Risk	Vul.
1	Null check + RuntimeException + Alert	0	✗
2	Null check + CameraAccessException + Alert	0	✗
3	hasSystemFeature + RuntimeEx. + Alert	0	✗
4	hasSystemFeature + CameraAccessEx. + Alert	0	✗
5	Null check + Alert	1	✓
6	hasSystemFeature + Alert	1	✓
7	RuntimeException + Alert	1	✓
8	CameraAccessException + Alert	1	✓
9	Exception + Alert	2	✓
10	Null check + RuntimeException	3	✓
11	Null check + CameraAccessException	3	✓
12	hasSystemFeature + RuntimeException	3	✓
13	hasSystemFeature + CameraAccessException	3	✓
14	Null check	4	✓
15	hasSystemFeature	4	✓
16	RuntimeException	4	✓
17	CameraAccessException	4	✓
18	Exception	4	✓
19	Nil	5	✓

Compared with cameras, the use of touchscreen is relatively straightforward. That is, an app can only be able to passively react to touchscreen events. If there is any Pre-Use attacks on touchscreen, there will be a window above this app during its inputting session (at least for our validated procedure in Section IV). So, the criterion to answer RQ3 is:

Criterion C3 for answering RQ3. During inputting session of sensitive information, if an app does not verify whether its window is obscured by another window, the app is vulnerable to Pre-Use attacks on touchscreens.

As demonstrated in Section IV, without learning about other windows, Pre-Use attacks can easily suffer from Pre-Use attacks.

Hence, a protection against Pre-Use attacks on touchscreen is to check any existence of other windows on receiving touch events. Android offers an API (`getFlags()` and `FLAG_WINDOW_IS_OBSCURED`) to indicate whether any window exists when a touch event occurs.

B. Design of RRACHECK

RRACHECK is designed to directly work on apk files. Given an apk file, RRACHECK transforms it into the *Jimple* (an intermediate language) representation used in Soot [10]. Soot is a widely adopted framework for Java program optimization [26]. For the implementation, RRACHECK is built on top of FlowDroid [10], where FlowDroid is built atop Soot. RRACHECK constructs an abstract syntax tree (AST) and builds up a method call graph (MCG). An inter-procedure control flow graph (ICFG) is also built from MCG (e.g., to determine how an exception in a call is handled in different methods). For any statement invoking APIs related to cameras or touchscreen, RRACHECK traverses the MCG to collect all methods calling any of these APIs.

RRACHECK also constructs a data dependency graph (DDG) consisting of data dependencies for two statements. The data dependency of two statements are defined as: if a variable v is defined in a statement s_1 and is used in another statement s_2, then s_2 has a data dependency on s_1. As Android is an event-driven system, components (e.g., `Activity`) are communicated through sending intents. When building the DDG, we leverage IccTA [27] to identify the receiver of each intent. The DDG is used to check whether an app checks a data against another data (e.g., `null` checking).

Next, we implemented three flows to check three criteria (C1, C2, and C3), as shown in Figure 7. We have verified our implementation on 50 open-source apps from `f-droid.com`. We performed a manual examination on these apps from their source code. Based on the three criteria, our manual analysis result was exactly the same as that produced by RRACHECK.

C. Results: Pre-Use Attack Analyses on Camera

Based on our analysis, there were 672 out of 1,000 apps requesting `CAMERA` permissions. And 337 of them actually create camera objects but 285 apps send intent with the action `MediaStore.ACTION_IMAGE_CAPTURE`, `ACTION_IMAGE_CAPTURE_SECURE`, or `ACTION_VIDEO_CAPTURE` to other apps and leverage them to take picture or record video. 121 apps only call `MediaRecorder.start()` to record video and they do not call APIs to take pictures. Moreover, 238 apps neither call camera related APIs nor send out camera related intent.

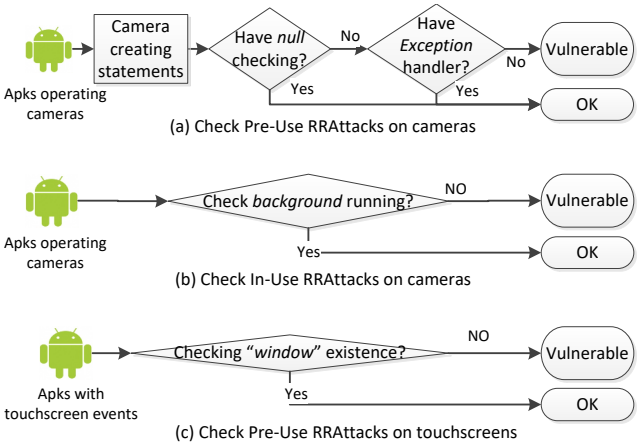

Fig. 7. Flows to Check Three RRAttacks.

Among the 337 apps creating camera objects (in 1,030 methods), 247 apps either check camera objects against `null` (in 603 methods) or capture exception via `try-catch` blocks (in 530 methods). That is, according to the criteria for answering **RQ1**, 90 (90/337=26.7%) apps are regarded as vulnerable to Pre-Use attacks on cameras.

We analyzed the protection mechanisms adopted in the 337 apps. Only one app registers a callback (i.e., call `registerAvailabilityCallback()`) to be notified on camera device availability (i.e., `com.wiscomwis.facetoface`). On creating `null` camera objects or capturing any exceptions, 63 apps either record the event in log (for 48 apps by calling `Log.e()` but without notifying users) or notify users (for 15 apps via Toast or Popup alert dialog). The remaining 273 apps out of the 337 apps neither record anything in log nor notify users on failing to create camera objects.

We further analyzed whether any apps will notify users about any failure after checking camera objects against `null` or capturing exceptions. The result shows that 6 apps do so. In details, 4 apps notify users on creating a `null` camera object and 2 apps notify users on capturing either `Exception` (i.e., `com.camera.comx.pic`) or `RuntimeException` (i.e., `com.example.anchortooldemo`). Additionally, 9 apps both check camera objects against `null` and capture exceptions but without notifying users.

Answer to RQ1: Among 337 apps creating camera objects out of the 1,000 apps, 90 apps (90/337=26.7%) are vulnerable to Pre-Use attacks on cameras. And 273 apps (273/337=81.0%) neither notify users or record anything in log on failing to create camera objects.

Among the 337 apps creating camera objects, there are 233 of them calling `ActivityManager.getRunningAppProcesses()` and comparing the result

with one of the three flags `IMPORTANCE_FOREGROUND`, `IMPORTANCE_CHANGED`, or `IMPORTANCE_BACKGROUND`. That is, according to the criteria for answering **RQ2**, 104 apps (104/337=30.9%) are vulnerable to In-Use attacks on cameras.

On checking protections by these apps, no app notify users when they are moved to background. (Note that, some apps indeed have notifications to users; however, all of these notifications are related to photo uploading, file scanning, or optimizations that are not related to their running status.)

Answer to RQ2: Among 337 apps creating camera objects out of the 1,000 apps, 104 apps (104/337=30.9%) are vulnerable to In-Use attacks on cameras. And all apps (100%) take no mechanism when they are moved into background.

D. Result: Pre-Use Attacks on Touchscreen

RQ3 is on the Pre-Use attacks on touchscreen to steal inputs from users. Hence, we limit our analysis to apps that have confidential inputs (like passwords). In this study, we only treat passwords in login activities as confidential inputs. Hence, we manually examined all apps to identify all apps with Login activities. This results in 244 apps out of the 1,000 apps. Next, we passed these apps to our RRACHECK.

Among the 244 checked apps, only 20 of them call API `MotionEvent.getFlags()`. However, only one of these 20 apps actually checks the flag `FLAG_WINDOW_IS_OBSCURED` (i.e., the method `onFilterTouchEventForSecurity()`). We have ensured the correctness of our RRACHECK by randomly selecting 20 apps and manually inspecting their source code. YouTube is the only app (`com.google.android.youtube`).

Surprisingly, the result indicates that, according to the criteria for answering **RQ3**, 243 apps (243/244=99.6%) are vulnerable to Pre-Use attacks on touchscreen. Only one app seems to take some mechanism against this kind of attacks. However, our further inspection into this app shows that, although it checks the flag, it does not take any mechanism like notifying users.

Answer to RQ3: Among 244 apps requiring passwords out of the 1,000 apps, 243 (243/244=99.6%) are vulnerable to Pre-Use attacks on touchscreen. No protection is found in these apps.

Discussion. Our above analysis only measures a lower bound. This is because, even if an app contains code to handle some "unusual case", they may still be vulnerable to RRAttacks. For example, when an app cannot open a camera for the first time, it may try for more times before notifying users. During this period, a RRAttack might have been done. We have encountered such a case on WeChat: it waits for 5 seconds before notifying camera unavailability. From our experiment, a 5-seconds period is enough to launch both Pre-Use and Post-Use attacks on cameras.

TABLE IV
SUGGESTED DEFENSE STRATEGY AGAINST RRATTACKS.

		Suggested Defense Strategy
Camera	Pre-Use	**[User App]** Check camera availability first and issue warnings if unavailable.
	In-Use	**[User App]** Issue warnings if the focus of an opened camera is lost. **[Sys. App]** Ensure itself is running in the foreground while using cameras.
	Post-Use	**[User App]** Hold a camera until the resource is consumed (e.g., A QR code becomes invalid; or a target becomes invisible to camera).
TouchScreen	Pre-Use	**[User App]** Check `FLAG_WINDOW_IS_OBSCURED` flag in touch events to determine any malicious windows. Or for sensitive views, set the `filterTouchesWhenObscured` attribute to ignore touch events dispatched from other windows. **[Sys. App]** Ensure no other window existed above it. **[Android]** Provide an *Interrupted*-like flag in each event to indicate whether any touch has been received by a third app in between two consecutive touches received by the current app.

VI. DEFENSE OF RRATTACKS

It is difficult for users to well manage resource permission on Android to defend RRAttacks. Android apps should also be equipped to be aware of and to prevent RRAttacks. We present several defense strategies on cameras and touchscreens, as listed in Table IV.

A. Defend RRAttacks on Cameras

Cameras suffer from all three attacks. We assume that, when a user app requests a camera, a camera is expected to be available. Hence, an app should firstly check the availability of a camera and immediately issue a warning if it is unavailable.

Hence, to prevent In-Use and Pre-Use attacks on cameras, an app should immediately issue a warning if it lost an opened camera. For system apps, they should make sure that they have the highest priority such that the opened camera cannot be preempted by other apps. Developers can still use the Camera2 API to protect against the attacks. By registering a listener on the camera's availability (`CameraManager.AvailabilityCallback`), attacked apps will be informed that they lost the control of the camera in use (or requested). Then, the attacked app can return to foreground by starting an activity to inform users that there is a potential malicious app.

To prevent Post-Use attacks on cameras, an app should hold camera for some time after taking photos to ensure that the involved environment is consumed (e.g., any target has been moved out of scope of cameras) or is no longer valid (e.g., for a QR code).

Discussion. Our above suggestion actually contradicts the widely suggested strategy: an app should release any resource as early as possible to avoid resource leaks [5], [28], which is also a common practice: *"Once your application is done using the camera, it's time to clean up"* [29]. However, from the perspective of security, the common practice may be harmful considering Post-Use attacks.

B. Defend RRAttacks on Touchscreen

Touchscreen only suffers from Pre-Use attacks. We suggest an app check the flag `FLAG_WINDOW_IS_OBSCURED` to determine whether a window may exist above it. Or it should filter touch events dispatched from other windows by setting `filterTouchesWhenObscured`. System apps can execute `dumpsys window displays` to check whether any window exists.

For Pre-Use attacks on touchscreen by trapping user inputs (see Section IV-C), there is no good way to allow an underlying user app be notified. Tracking user input speed might be an option. Another possible way is, for Android system, to provide an `Interrupted`-like flag within each event. This flag can be used to check that, whether any touch has been received by a third app (excluding Android system) in between two consecutive touches received by a user app.

VII. Related Work

A. Discussion on Related Works

The work [30] relies on a two-phase (i.e., training and testing) machine learning based approach to determine when to launch the attack; whereas RRAttack does not need such a heavy-weight approach. Take RRAttack on camera as an example. RRAttack has three attacks on camera (Pre/In/Post) whereas the latter has only one (which is a special case of our Post-Use RRAttack) and it cannot launch both Pre-Use attacks and In-Use attacks. In details, the latter frequently requests a camera in background to infer whether it is released (if so, it takes photos). In contrast, our Post-Use RRAttacks is much more efficient because it passively listens to whether a camera is just released (if so, it takes photos).

Besides, Google's latest policy renders the attack in [30] ineffective because taking photos at background is forbidden since Android P [17]. However, such a policy does not affect RRAttack as explained in this paper.

The work [31] is a kind of Task-Hijacking attacks; whereas RRAttack does not rely on task-hijacking. Take RRAttack on camera as an example. When a camera is in-use or is just released, we launch a new activity with flag `FLAG_ACTIVITY_NEW_TASK` from a background service. Therefore, the launched activity is placed in a new task stack instead of the one used by the victim app.

B. Other Related Works

In traditional desktop and browser environments, user interface attacks and side channel attacks have been well studied [32]–[43]. With the ubiquity of mobile devices especially the Android smartphones, these attacks have been explored by researchers and malicious adversaries to mobile environments.

GUI Attacks on Android. A variety of Android GUI attacks have been investigated by researchers. Niemietz et al. [44] ported desktop-based UI redressing attacks, including the classic clickjacking attack and the tapjacking attack, to Android devices. These attacks involve two participants, a visible attacker's foreground app in form of a notification and a target background App. They can be seen as the primary window overlay attacks (or draw-on-top attacks). Roesner et al. [45] elaborated a wide range of GUI related attacks involving embedded (or child) user interfaces and host (or parent) view components. For instance, the display forgery attack means the parent App modifies the child element, the input forgery attack denotes that the host App delivers the forged user input into an embedded view component, and the DoS attack refers that the parent App prevents user input from reaching a child element. Bianchi et al. [46] proposed a series of GUI confusion attacks to exploit the user's inability to verify which App is drawing on the screen and receiving user events.

Side Channel Attacks on Android. Since the smartphone consists of a plethora of embedded features and sensors, a growing number of side channel attacks targeting Android devices have been proposed [47]. Traditionally, attackers exploit information leaked from physical side channels to infer personal or industrial sensitive information. Cai et al. [48] proposed a keystroke inference attack based on the exploration that keystroke vibrations are highly correlated to the keys being typed as demonstrated. Das et al. [49] simultaneously used both of the accelerometer and the gyroscope to produce the accurate device fingerprint. The camera and the microphone are two other commonly misused side channels to infer users' sensitive events or physical surroundings. Templeman et al. [50] introduced a novel visual malware that allows remote hackers to reconstruct rich three-dimensional models of the user's personal indoor spaces. Simon et al. [51] described a novel side channel attack on camera and microphone to infer PINs entered on a number-only soft keyboard. Fiebig et al. [52] analyzed images of facial reflections captured by the front-facing camera, to infer users' keystrokes. Narain et al. [53] designed a sound trojan that records users' dialogs via the microphone to further recognize users' credit card numbers through audio processing algorithms.

VIII. Conclusion

We present Resource Race attacks (RRAttacks) on Android and identify that both cameras and touchscreen are vulnerable to RRAttacks including both POC attacks and real-world attacks. We further report an experiment over 1,000 apps on whether they are vulnerable to RRAttacks. Based on the causes of RRAttacks, we propose several strategies on defense of RRAttacks for user apps, system apps, and Android system.

Acknowledgment

This work is supported in part by the National Natural Science Foundation of China (NSFC) (Grant No. U1736209, 61572483, 61602457, and 61932012), the Key Research Program of Frontier Sciences, CAS (Grant No. ZDBS-LY-7006 and QYZDJ-SSW-JSC036), the Youth Innovation Promotion Association of the Chinese Academy of Sciences (YICAS) (Grant No. 2017151), the Young Elite Scientists Sponsorship Program by CAST (Grant No. 2017QNRC001), and the Hong Kong RGC Project (No. 152223/17E, CityU C1008-16G).

REFERENCES

[1] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang, "Characterizing and detecting resource leaks in android applications," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013, pp. 389–398.

[2] 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 *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 1041–1057.

[3] 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 Conference on Security Symposium*, ser. SEC'14, 2014, pp. 1037–1052.

[4] X. Bai, Z. Zhou, X. Wang, Z. Li, X. Mi, N. Zhang, T. Li, S.-M. Hu, and K. Zhang, "Picking up my tab: Understanding and mitigating synchronized token lifting and spending in mobile payment," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 593–608.

[5] T. Wu, J. Liu, Z. Xu, C. Guo, Y. Zhang, J. Yan, and J. Zhang, "Light-weight, inter-procedural and callback-aware resource leak detection for android apps," *IEEE Transactions on Software Engineering*, vol. 42, no. 11, pp. 1054–1076, 2016.

[6] Y. Z. X. Jiang and Z. Xuxian, "Detecting passive content leaks and pollution in android applications," in *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS)*, 2013.

[7] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 229–240.

[8] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt, "Identity, location, disease and more: Inferring your secrets from android public resources," in *Proceedings of the 2013 ACM SIGSAC conference on Computer and communications security*, 2013, pp. 1017–1028.

[9] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: detecting malicious apps in official and alternative android markets." in *NDSS*, vol. 25, no. 4, 2012, pp. 50–52.

[10] 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*, vol. 49, no. 6, pp. 259–269, 2014.

[11] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Using probabilistic generative models for ranking risks of android apps," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 241–252.

[12] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proceedings of the second ACM conference on Data and Application Security and Privacy*, 2012, pp. 317–326.

[13] "Race condition," https://en.wikipedia.org/wiki/Race_condition, (Last accessed on June, 2019).

[14] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Transactions on Computer Systems (TOCS)*, vol. 15, no. 4, pp. 391–411, 1997.

[15] Y. Cai, B. Zhu, R. Meng, H. Yun, L. He, P. Su, and B. Liang, "Detecting concurrency memory corruption vulnerabilities," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: ACM, 2019, pp. 706–717.

[16] Google, "Android camera2 overview," https://developer.android.com/reference/android/hardware/camera2/package-summary.html, (Last accessed on June, 2019).

[17] "A boon for privacy: Android p will prevent idle background apps from accessing the camera," https://www.xda-developers.com/android-p-background-apps-camera/, (Last accessed on June, 2019).

[18] K. Kennedy, E. Gustafson, and H. Chen, "Quantifying the effects of removing permissions from android applications," in *Workshop on Mobile Security Technologies (MoST)*, 2013.

[19] "Qr code reader," https://play.google.com/store/apps/details?id=tw.mobileapp.qrcode.banner, (Last accessed on June, 2019).

[20] "Ocr," https://app.xunjiepdf.com/en/ocr, (Last accessed on June, 2019).

[21] "Activity," https://developer.android.com/reference/android/app/Activity, 2018.

[22] "Remove window obscurement information," https://android.googlesource.com/platform/frameworks/native/+/5508ca2c191f8fdf29d8898890a58bf1a3a225b3, 2018.

[23] Z. Xu, K. Bai, and S. Zhu, "Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors," in *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WISEC '12, 2012, pp. 113–124.

[24] A. Ghosh and G. Riccardi, "Recognizing human activities from smartphone sensor signals," in *Proceedings of the 22Nd ACM International Conference on Multimedia*, ser. MM '14. New York, NY, USA: ACM, 2014, pp. 865–868.

[25] K. Vertanen, H. Memmi, J. Emge, S. Reyal, and P. O. Kristensson, "Velocitap: Investigating fast mobile text entry using sentence-based decoding of touchscreen keyboard input," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, ser. CHI '15. New York, NY, USA: ACM, 2015, pp. 659–668.

[26] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, ser. CASCON '10, 2010, pp. 214–224.

[27] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *Proc. ICSE*, 2015.

[28] Y. Liu, C. Xu, S. C. Cheung, and J. LÃij, "Greendroid: Automated diagnosis of energy inefficiency for smartphone applications," *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 911–940, Sept 2014.

[29] "Control the camera," https://developer.android.com/training/camera/cameradirect, 2018.

[30] 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*, ser. SEC '14, 2014, pp. 1037–1052.

[31] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu, "Towards discovering and understanding task hijacking in android," in *Proceedings of the 24th USENIX Security Symposium*, ser. SEC '15, Washington, D.C., 2015, pp. 945–959.

[32] S. Chen, J. Meseguer, R. Sasse, H. J. Wang, and Y.-M. Wang, "A systematic approach to uncover security flaws in gui logic," in *Proceedings of the 28th IEEE Symposium on Security and Privacy*, ser. SP '07, 2007, pp. 71–85.

[33] D. Akhawe, W. He, Z. Li, R. Moazzezi, and D. Song, "Clickjacking revisited: A perceptual view of ui security," in *Proceedings of the 8th USENIX Workshop on Offensive Technologies*, 2014.

[34] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schecter, and C. Jackson, "Clickjacking: Attacks and defenses," in *Proceedings of the 21st USENIX Security Symposium*, ser. SEC '12. USENIX, 2012, pp. 413–428.

[35] S. Lekies, M. Heiderich, D. Appelt, T. Holz, and M. Johns, "On the fragility and limitations of current browser-provided clickjacking protection schemes," in *Proceedings of the 6th USENIX Workshop on Offensive Technologies*. USENIX, 2012.

[36] S. Duman, K. Onarlioglu, A. O. Ulusoy, W. Robertson, and E. Kirda, "Trueclick: Automatically distinguishing trick banners from genuine download links," in *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014, pp. 456–465.

[37] M. Johns and S. Lekies, "Tamper-resistant likejacking protection," in *Proceedings of the 16th International Symposium on Research in Attacks, Intrusions and Defenses*, 2013, pp. 265–285.

[38] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *Proceedings of the 30th IEEE Symposium on Security and Privacy*, ser. SP '09, 2009, pp. 45–60.

[39] S. Chen, R. Wang, X. Wang, and K. Zhang, "Side-channel leaks in web applications: A reality today, a challenge tomorrow," in *Proceedings of the 31st IEEE Symposium on Security and Privacy*, ser. SP '10, 2010, pp. 191–206.

[40] Z. Weinberg, E. Y. Chen, P. R. Jayaraman, and C. Jackson, "I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks," in *Proceedings of the 32nd IEEE Symposium on Security and Privacy*, ser. SP '11, 2011, pp. 147–161.

[41] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Proceedings of the 36th IEEE Symposium on Security and Privacy*, ser. SP '15, 2015, pp. 605–622.

[42] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *Proceedings of the 19th ACM Conference on Computer and Communications Security*, ser. CCS '12, 2012, pp. 305–316.

[43] R. Callan, A. Zajic, and M. Prvulovic, "A practical methodology for measuring the side-channel signal available to the attacker for instruction-level events," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '14, 2014, pp. 242–254.

[44] "Ui redressing attacks on android devices," https://media.blackhat.com/ad-12/Niemietz/bh-ad-12-androidmarcus_niemietz-WP.pdf, 2012.

[45] F. Roesner and T. Kohno, "Securing embedded user interfaces: Android and beyond," in *Proceedings of the 22nd USENIX Security Symposium*, ser. SEC '13, Washington, D.C., 2013, pp. 97–112.

[46] 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 36th IEEE Symposium on Security and Privacy*, ser. SP '15, 2015, pp. 931–948.

[47] R. Spreitzer, V. Moonsamy, T. Korak, and S. Mangard, "Systematic classification of side-channel attacks: A case study for mobile devices," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 1, pp. 465–488, 2018.

[48] L. Cai and H. Chen, "Touchlogger: Inferring keystrokes on touch screen from smartphone motion," in *Proceedings of the 6th USENIX Conference on Hot Topics in Security*, 2011, pp. 9–9.

[49] A. Das, N. Borisov, and M. Caesar, "Tracking mobile web users through motion sensors: Attacks and defenses," in *Proceedings of the 23rd Annual Network & Distributed System Security Symposium*, 2016.

[50] R. Templeman, Z. Rahman, D. Crandall, and A. Kapadia, "Placeraider: Virtual theft in physical spaces with smartphones," in *Proceedings of the 20th Annual Network & Distributed System Security Symposium*, 2013.

[51] L. Simon and R. Anderson, "Pin skimmer: Inferring pins through the camera and microphone," in *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, ser. SPSM '13, 2013, pp. 67–78.

[52] T. Fiebig, J. Krissler, and R. Hänsch, "Security impact of high resolution smartphone cameras," in *Proceedgins of the 8th USENIX Workshop on Offensive Technologies*, ser. WOOT '14, 2014.

[53] R. Schlegel, K. Zhang, X.-y. Zhou, M. Intwala, A. Kapadia, and X. Wang, "Soundcomber: A stealthy and context-aware sound trojan for smartphones." in *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, ser. NDSS '11, 2011, pp. 17–33.