See No Evil: Phishing for Permissions with False Transparency

Güliz Seray Tuncay, *Google, University of Illinois at Urbana-Champaign;*
Jingyu Qian and Carl A. Gunter, *University of Illinois at Urbana-Champaign*

https://www.usenix.org/conference/usenixsecurity20/presentation/tuncay

This paper is included in the Proceedings of the
29th USENIX Security Symposium.

August 12–14, 2020

978-1-939133-17-5

See No Evil: Phishing for Permissions with False Transparency

Güliz Seray Tuncay
Google, University of Illinois at Urbana-Champaign
gulizseray@google.com

Jingyu Qian
University of Illinois at Urbana-Champaign
jingyuq2@illinois.edu

Carl A. Gunter
University of Illinois at Urbana-Champaign
cgunter@illinois.edu

Abstract

Android introduced runtime permissions in order to provide users with more contextual information to make informed decisions as well as with finer granularity when dealing with permissions. In this work, we identified that the correct operation of the runtime permission model relies on certain implicit assumptions which can conveniently be broken by adversaries to illegitimately obtain permissions from the background while impersonating foreground apps. We call this detrimental scenario *false transparency attacks*. These attacks constitute a serious security threat to the Android platform as they invalidate the security guarantees of 1) runtime permissions by enabling background apps to spoof the context and identity of foreground apps when requesting permissions and of 2) Android permissions altogether by allowing adversaries to exploit users' trust in other apps to obtain permissions.

We demonstrated via a user study we conducted on Amazon Mechanical Turk that mobile users' comprehension of runtime permissions renders them susceptible to this attack vector. We carefully designed our attacks to launch strategically in order to appear persuasive and verified the validity of our design strategies through our user study. To demonstrate the feasibility of our attacks, we conducted an in-lab user study in a realistic setting and showed that none of the subjects noticed our attacks. Finally, we discuss why the existing defenses against mobile phishing fail in the context of false transparency attacks. In particular, we disclose the security vulnerabilities we identified in a key security mechanism added in Android 10. We then propose a list of countermeasures to be implemented on the Android platform and on app stores to practically tackle false transparency attacks.

1 Introduction

Be transparent. When you make a permissions request, be clear about what you're accessing, and why, so users can make informed decisions. – App permissions best practices by Google [1]

Android's permission system enables access control on sensitive user data and platform resources based on user consent. In an effort to foster meaningful consent, Android 6 introduced runtime permissions to help users understand why a permission is needed by an app by asking for it in a relevant use context. In particular, the runtime permission model warrants certain security guarantees to achieve this goal. First, it provides a *contextual guarantee* to ensure that users will always be given the necessary contextual information to make informed decisions, by enforcing apps to be in the foreground when requesting permissions. Second, it provides an *identity guarantee* to ensure that users are well-aware of the identity of the app owning the current context during a permission request, by clearly displaying the name of the requesting app in permission dialogs. In line with its ultimate goal of providing context, the model also relies on the cooperation of app developers to be transparent regarding their need of permissions during permission requests.

In this work, we have identified that the security guarantees of runtime permissions are broken due to some implicit assumptions made by the platform designers. To this end, we show how our findings can be used by adversaries to stealthily obtain runtime permissions. First, Android assumes that an app in the foreground will *always* present a meaningful and legitimate context. However, we show that background apps can surreptitiously move to the foreground to request permissions without being noticed by users by utilizing Android APIs and invisible graphical displays. Second, we observed that the naming scheme utilized in the permission dialogs assumes that the app name will uniquely identify a single app installed on the user device for its secure operation; however, the Android ecosystem does not readily enforce any rules on these names. As a consequence, apps can spoof the names of other apps or use irrelevant names that have the potential to mislead users regarding the true source of the request. The combination of these findings indicate the possibility of phishing-based privilege escalation on runtime permissions; a background app (adversary) can now request and obtain permissions while leading the user to believe the request was triggered by the foreground app (victim), an insidious phishing scenario which we call *false transparency attacks* on runtime permissions.

In false transparency attacks, a permission request from an app is not transparent in the sense of the quote above. Instead, its context is literally transparent as a *graphical display* to ensure that the user only sees the victim app at the time of the request. In addition, its origin is intentionally set to mislead the user into thinking the request was triggered by the foreground app. Hence, the adversary can take advantage of both the context and the identity of a victim app for its own requests to more conveniently obtain permissions. To illustrate, suppose Vibr is an attack app that launches a false transparency attack from the background while the widely-used communication app Viber is in the foreground. If, for example, Vibr requests permission to access the user's contacts, then an unwary user may grant this if they think the request comes from Viber, as users would generally feel comfortable granting contacts permission to a popular communication app that needs it for its utility [2]. In particular, we argue that these attacks constitute a notable security hazard for the Android platform for two reasons. First, they allow background apps to stealthily obtain permissions while providing a *spoofed* context and identity, which clearly defeats the purpose of using runtime permissions to provide meaningful contextual information to users. Second, they create an opportunity for malicious apps to exploit the user's trust in another, possibly high-profile app to obtain permissions that they would normally not be able to acquire, breaking the security guarantees of Android permissions altogether.

False transparency attacks serve as a platform for adversaries to obtain *any* set of dangerous permissions by exploiting user's trust in other apps. In order to profitably mislead users to grant permissions, the permission dialogs triggered by adversaries should appear plausible, justifying the need for a permission, as users have a strong tendency towards denying requests that seem irrelevant to the app's use [2, 3]. We first show via a user study we conduct on Amazon Mechanical Turk that users indeed demonstrate susceptibility to false transparency attacks due to a lack of complete understanding of the runtime permission model (e.g., security guarantees of runtime permissions). We then propose various key schemes to launch our attacks strategically and implement them after verifying with our user study that they would indeed lead to more stealthy and effective attacks. Furthermore, we conduct an in-lab user study in a realistic setting to verify the feasibility of our attacks and show that none of the subjects noticed they had been attacked.

Additionally, we study the existing defense mechanisms against mobile phishing and discuss why they fall short in the context of false transparency attacks. A noteworthy one among these defenses is the strategy recently introduced by Google in Android 10. We have found that this security mechanism suffers from serious security vulnerabilities and design issues, which rendered our attacks still effective on this Android version and onward. Finally, we propose a list of countermeasures that can be practically implemented on the Android platform and on app stores such as Google Play to defend against false transparency attacks.

Our contributions can be summarized as follows:

- We uncovered design shortcomings in Android's runtime permissions which inadvertently lead to a violation of the essential security guarantees of this permission model.
- By utilizing these shortcomings, we built false transparency attacks, which enable adversaries to illegitimately obtain permissions using a victim app's context and identity.
- We conducted a user study to understand if users' comprehension of runtime permissions created susceptibility to this attack vector as well as to verify the validity of our design strategies for stealthy attacks.
- We conducted a user study to demonstrate the feasibility of our attacks in a realistic setting.
- We discovered serious issues in the new security mechanism that addresses phishing in Android 10 and later and showed the feasibility of our attacks on these versions.
- We proposed practical countermeasures that can effectively tackle false transparency attacks.

2 Background

2.1 Android Permissions

Previously, permissions were permanently granted to apps at installation time on Android. With the introduction of Android 6.0 (API level 23), Android switched to runtime permissions where permissions for high-risk resources (e.g., camera, contacts etc.) are requested dynamically and could be revoked by the user at any time. This was done in an effort to provide users more context while making decisions [4]. In this permission model, users are presented with a permission dialog on or before the first use of a sensitive resource that is protected with a dangerous permission and are given the ability to allow or deny a permission request. Furthermore, users can adjust app permissions at any time through the system settings.

The `PackageManager` class can be queried to obtain permission information of apps. In particular, the `getInstalledPackages()` API can be used with the `PackageManager.GET_PERMISSIONS` flag to obtain the permissions requested by apps as stated in their manifests and the current states of these permissions (i.e., granted or not).

2.2 App Components and Task Organization

Apps can contain four main components on Android: activities, services, broadcast receivers, and content providers. An `Activity` presents the user with a single-purpose user interface. A `Fragment` is an activity subcomponent that is utilized to represent certain behavior or a portion of UI. A `Service` performs long-running tasks in the background. A `BroadcastReceiver` enables receiving messages from other apps or the system. Finally, a `ContentProvider` provides apps with a relational database interface to enable storing and sharing data. Android provides the `Intent` messaging scheme as a part of its Binder IPC mechanism to enable communication between these components.

On Android, a task is a collection of activities that collaboratively perform a specific job. Android arranges activities forming a task into a stack, in the reverse order of their initiation. Pressing the back button removes the top activity from the stack and brings forth the activity underneath to the foreground. In addition, recently-accessed tasks can be obtained via clicking the recents button to view a system-level UI called the recents screen. Normally, the system handles the addition and removal of activities and tasks to/from the recents screen; however, this behavior can be overridden. For instance, tasks can be excluded by setting `android:excludeFromRecents` or by calling the `finishAndRemoveTask()` API in the activity creating the task.

3 Runtime Permissions in the Wild

Our attacks constitute a notable threat to the security of runtime permissions. Here, we study the adoption of runtime permissions to demonstrate the extent of our attacks. First, we investigate the adoption of Android versions that support runtime permissions (Android 6-11). As reported by Google, the cumulative adoption of these Android versions is 74.8% [5]. This means that the majority of users are using Android devices that support runtime permissions and are vulnerable to our attacks by default. Next, we investigate the prevalence of apps that adopted runtime permissions. For this purpose, we collected the 80 top free apps of each app category on Google Play (with some failures) and obtained a final dataset with 2483 apps. We collected this dataset in December 2018, when runtime permissions had already been released for a few years. Table 1 summarizes our results. 83% of the apps in our dataset have a target API level 23 or more, indicating they utilize runtime permissions. Out of these apps, 85% of them (71% of all apps) request at least one permission of protection level dangerous. This shows that runtime permissions are highly adopted by app developers and users are already accustomed to dealing with runtime permissions as the majority of the apps request permissions at runtime.

4 Attacking Runtime Permissions

Android's runtime permission model provides essential security guarantees in order to reliably and securely deliver contextual information. In this section, we will discuss these guarantees and explain how they can be broken to launch phishing-based privilege escalation attacks that we call *false transparency attacks* on runtime permissions. We will then discuss the internals of our attacks in detail.

Table 1: Adoption of permission models and use of dangerous permissions by apps (# of apps (% of apps)).

Requesting dangerous permissions?	Using Runtime permissions	Using Install-time permissions
Yes	1755 (71%)	357 (14%)
No	309 (13%)	62 (2%)
Total	**2064 (83%)**	**419 (17%)**

Threat model. We assume an adversary that can build Android apps and distribute them on app markets, such as Google Play (GP) store; however, the adversary is not necessarily a reputable developer on GP. They provide an app with some simple and seemingly useful functionality (e.g., QR code scanner etc.) to lure the users into installing the app. Their goal is to obtain a desired set of dangerous permissions, which is relatively difficult to achieve for non-reputable app developers and is especially harder if their app does not have a convincing reason to why it requires a specific permission [2].

4.1 (Breaking) the Security Guarantees of Runtime Permissions

Runtime permissions strive to provide contextual information to users to help them make conscious decisions while granting or denying permissions. In order to reliably and securely deliver this contextual information, Android warrants some security guarantees: 1) users will always be provided with the necessary contextual information, 2) users will be informed about the identity of the app owning the current context at the time of a permission request. Here, we discuss how these guarantees rely on the validity of certain implicit assumptions and present ways to invalidate them, undermining the security of runtime permissions.

Contextual guarantee. This security guarantee states that *users should always be provided with context during requests.* Android attempts to achieve this by allowing apps to request permissions *only* from the foreground. The runtime permission model aims to provide users with increased situational context to help them with their permission decisions. Currently, Android provides contextual information to dynamic permission requests in the form of graphical user interfaces. That is, when the user is presented with a permission dialog, they have the knowledge of what was on the screen and what they were doing prior to the request to help them understand how this permission might be used by the app. In order to ensure users are *always* provided with contextual information at the time of permission requests, Android allows permissions to be requested *only* from the context of UI-based app components such as activities and fragments and the requesting component has to be in the foreground at the time of the request. The assumption here is that since apps are allowed to request permissions only from the foreground, users will *always* be provided with a *meaningful* context. In our work, we show that this assumption is conceptually broken as we can utilize the existing features offered to developers by the Android platform to enable background apps to stealthily request permissions from illegitimate contexts. To elaborate, Android provides mechanisms that give apps the ability to move within the activity stack. In addition, activities can be made *transparent*, simply by setting a translucent theme. By combining both of these mechanisms, a transparent background app can be surreptitiously brought to the foreground for a limited amount of time, only to immediately request a permission. Once the request is completed by the user, the app can be moved to the back of the task stack again. This way, a background app gains the ability to request permissions without providing any *real* context as the user is presented only with the context of the legitimate foreground app due to the transparency of the background app that is overlaid on top. It is important to note that permission requests freeze the UI thread of the requesting app so nothing will happen if the user clicks on the screen to interact with the app itself on Android $\leq$10. This way, users will not have the opportunity to detect the mismatch between the supposed and actual foreground app by simply trying to interact with the app. On Android 11, the permission dialog disappears if the user clicks somewhere else than the dialog itself. In this case, the adversarial app can simply move to the background following the user click.

Bianchi et al. discusses some of the ways they discovered how a background app can be moved to the foreground [6]. Here, we discuss some of these techniques that were previously discussed as well as some other ways we discovered that could achieve the same goal.

• *startActivity API.* Android provides the `startActivity` API to start new activities, as the name suggests. According to Bianchi et al., using this API to start an activity from a service, a broadcast receiver or a content provider will place the activity at the top of the stack if `NEW_TASK` flag is set. However, we found that simply calling `startActivity` without setting this flag in these components works similarly in recent Android versions. In addition, they found that starting an activity from another activity while setting the `singleInstance` flag also places the new activity on top of the stack. We found that setting this flag is not necessary to achieve this anymore, even simply starting an activity from a background activity seems to bring the app to the foreground.

• *moveTaskTo APIs.* `moveTaskToFront()` API can be used to bring an app to the foreground. This API requires the `REORDER_TASKS` permission, which is of normal protection level and is automatically granted at installation time. In addition, `moveTaskToBack()` API can be used to bring apps to the back of the task stack. In this case, we observed that the app continues to run in the background as `Activity.onStop()` is not called unless the activity actually finishes its job.

• *requestPermission API.* According to Android's official developer guides, `requestPermission(String[], int)` API can only be called from components with user interface such as activities and fragments. This is in line with the main goal of runtime permissions, to provide users a sense of situational context before they make decisions regarding permissions. A similar version of this API with different parameters is also implemented in the Android support APIs to provide forward-compatibility to legacy apps. This version, `requestPermission(Activity, String[], int)`, takes an extra activity parameter and requests the permission from the context of this activity. This support API makes it possible to request permissions from non-UI components, such as services and broadcast receivers. In addition, if this API is called from a non-UI component or from an activity running in the background, the app will be automatically brought to the foreground for the request on Android $\leq$ 9. On Android 10 and 11, this API does not bring background activities to the foreground.

Identity guarantee. According to this security guarantee, *users should be made aware of the identity of a requesting app.* Android attempts to achieve this by displaying the app's name in the permission dialog. Android allows apps to be started automatically via the intent mechanism for IPC without requiring user's approval or intervention. This can create an issue for permission requests since the user might not be able to tell the identity of an automatically-launched app if it were to immediately request a permission, as they have not personally started or been interacting with this app. In order to overcome this issue, Android displays the name of the requesting app in permission dialogs in order to help users quickly identify the app owning the current context.

Even though this mechanism initially seems like an effective solution to the app identification problem for runtime permissions, it is insufficient since app names are in fact not guaranteed by the Android ecosystem to uniquely identify apps on user devices. Each Android app listed on the Google Play (GP) store has a *Google Play listing name* that is displayed to the user while browsing this store, as well as an *internal app name* that is defined in the resource files of the app's apk and displayed when the user is interacting with the app on their device, including in the permission dialogs. Google Play does enforce certain restrictions on GP listing names. For example, it produces warnings to developers when their GP listing name is too similar to that of another app and does not allow the developers to publish their apps in this case, in an attempt to prevent typo-squatting and spoofing that can be used in phishing attacks. However, the same kind of scrutiny does not seem to be shown when it comes to internal app names as the Android ecosystem does not enforce any rules on these names. Our observation is that 1) the internal name of an app can be vastly different than the app's GP listing name and 2) multiple apps can share the same app name, even when installed on the same device. For example, we have successfully published an app on Google Play, where the internal name of our app was "this app" even though the GP listing name was completely different, a case we will make use of in our attacks as we will explain in more detail in the rest of this section. We were also able to spoof the name of a popular app (i.e., Viber) and successfully release our app with this app name on GP. In short, the Android ecosystem does not perform any verification on the app names shown in runtime permission dialogs to ensure their validity.

4.2 False Transparency Attacks

By combining the ability of apps to move within the task stack in disguise and Android's lack of app name verification, we built the *false transparency attacks*, where a *transparent* background app temporarily moves to the foreground while impersonating another, possibly more trustworthy app that was already in use by the user (i.e., in the foreground) and requests a permission it sees fit. After the user either responds to the permission request, the attack app immediately moves to the background again to evade detection so that the user can continue interacting with the legitimate foreground app without noticing they have been attacked. We verified that this is a general class of attacks that affects *all* Android versions that support runtime permissions (Android 6-11).

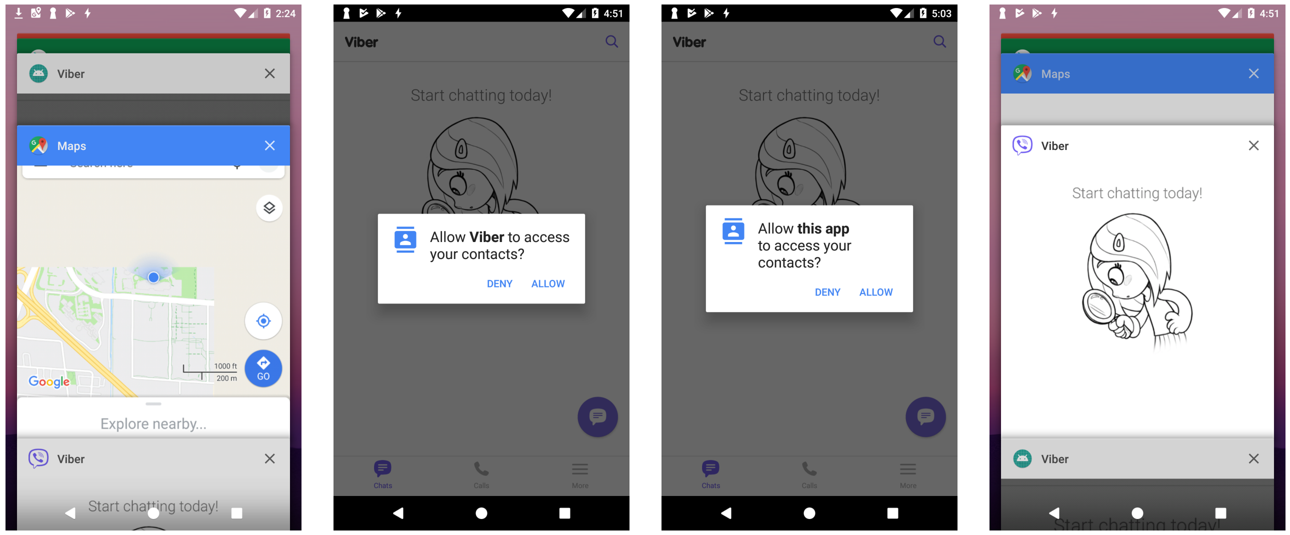
A demonstration of our attack including the state of the task stack before and during the attack can be observed in action in Figure 1. Figure 1a displays the task stack immediately before the attack takes place. As can be seen, Viber, a popular communication app with millions of downloads, is on the top of the task stack (shown in the bottom) and at the back of the task stack there is another app also called Viber, representing the attack app running in the background targeting Viber for permissions. Here, it is worth noting that we are showing the real content of the task stack for demonstration purposes and the attack app can in fact be easily hidden from the task stack in order to evade detection by the user, by utilizing the `finishAndRemoveTask()` API or the `android:excludeFromRecents` tag in the Android manifest file as discussed in Section 2.2.

At the time of the attack, the user will experience a user interface (UI) that is similar to the one in Figure 1b. Here, the app prompting the user for a permission appears to be Viber, as both the UI displayed underneath the permission dialog and the app name in the dialog indicate the app to be Viber. However, the request is, as a matter of fact, triggered from the transparent attack app that surreptitiously made its way to the foreground and covered Viber. This can be observed by displaying the state of the task stack at the time of the attack, as shown in Figure 1d. As can be seen, the forged Viber app that belongs to the attacker is in fact at the forefront of the task stack (seen at the bottom) and the real Viber app is immediately behind it at the time of the attack, creating a confusion about the origin of the permission request for users due to the identicalness of the shown user interface to that of Viber. Additionally, the attacker was able to spoof the internal app name of Viber in the permission dialog to further mislead the user into thinking the permission request indeed originated from Viber, as shown in Figure 1b. All in all, the contextual cues given to the user in this attack scenario (i.e., UI and app name) appear to be indistinguishable from a benign scenario where Viber is the legitimate app requesting a permission, from the perspective of device users.

We envision false transparency attacks to be useful for adversaries in two main scenarios. First, when users do not consider an app to be very trustworthy, they are much less likely to grant a permission, as shown by previous work [2]. Hence, an adversary without much reputation can utilize false transparency attacks to pose as a trusted app to obtain a permission. Second, in some cases, it might be nearly impossible for the adversary to provide a reasonable explanation to the user why their app might need a certain permission. For example, a malicious QR code app might also have the goal of getting user's contact list. The app can directly ask for the dangerous permission, but this may make the user suspicious: the user may deny the permission request or possibly even uninstall the app. In this case, false transparency attacks would give the adversary the opportunity to pose as a trusted app that is known to require this permission for its utility (e.g., Viber requiring contacts) without arousing suspicion.

Plausible and realistic attacks. We intend for our attacks to serve as a platform for adversaries to conveniently obtain *any* set of dangerous permissions by exploiting users' trust in other apps without arousing suspicion. With each request, the adversary is essentially exposing themselves to the user and is risking the possibility of alerting the user to be suspicious and take action accordingly (e.g., scan their apps to uninstall questionable ones). Therefore, it would be in the adversary's best interest to request permissions sparingly, only when the user is less likely to be alarmed due to the permission request. In order to achieve this, we utilize several strategies as we will now explain. We verify the validity of these strategies with a user study which we will present in detail in Section 6.

First, users are accustomed to being asked for permissions by an app running in the foreground under the runtime permission model. We show with our user study that users are indeed not very receptive of requests coming from no apparent foreground app. Hence, we do not request permissions when there is no app in the foreground. For this purpose, we utilize the `getRunningTasks()` API, which previously provided the identity (i.e., package name) of the app in the foreground, but was deprecated in Android 5 due to privacy reasons. However, we discovered that this API still provides limited amount



(a) Pre-attack task stack (b) Single-target attack (c) Multi-target attack (d) Task stack during the attack

Figure 1: Background app requesting a permission pretending to be the foreground app (Viber).

of information that can be utilized to detect the existence of a foreground app. More specifically, on Android 5-8 this API outputs `com.google.android.apps.nexuslauncher` if there is no app in the foreground, indicating the nexus launcher. Otherwise, it outputs the package name of the app calling the API (whether this app is in the foreground or not), indicating the existence of a running foreground app. Starting from Android 9, this API reports the most recent one between the nexus launcher and the caller app's own package name, again without revealing other foreground apps to the caller. In order to reliably use this information to detect the existence of an app in the foreground on Android 9 and later, the adversary first needs to briefly come to the foreground using one of the techniques described in Section 4.1 while using a transparent UI to evade detection then run `getRunningTasks()` after going back to the background. This ensures that adversary's app is *always* more recent than the nexus launcher when there is an app in the foreground.

Second, previous work has shown that when users make decisions to grant or deny permissions, they consider the relevance of a requested permission to the functionality of the app. If they think the app should not require a certain permission for its operation or it should still work without that permission, they generally choose to not grant the permission [2,3]. Taking this observation into account, in our attacks we avoid requesting permissions that are certainly irrelevant to the functionality of the foreground app because such requests will likely result in the user denying the permission. Here, we consider a permission to be relevant to the functionality of an app only if the app declares this permission in its manifest file and intends to use it. In order to identify these relevant permissions, we first need a mechanism to detect the identity of the victim app in the foreground (i.e., its package name) so that we can determine its declared permissions. For this purpose, we utilize ProcHarvester [7], which uses machine learning techniques to identify the public parts of Android's proc file system (procfs) that can be used to infer the foreground app on Android, even when access to procfs is mostly restricted by Google due to privacy reasons. We modify ProcHarvester to fit realistic attack scenarios and implement real-time inference of time series on top of ProcHarvester to detect the identity of the foreground app in real time. After obtaining the package name of the foreground app, we can use this information to query `PackageManager` to obtain the permissions required by this app and request only those permissions in our attacks. Section 2 explains how this information can be obtained. The details of our foreground app inference implementation will be described in Section 5.

Third, users' previous decisions in the context of permissions affect how the victim app behaves in terms of future permission requests. Hence, we argue that these decisions should also be taken into account by the attacker to avoid alarming the users. In particular, we argue that an attacker blindly requesting one of the permissions declared in a victim app's manifest file can still arouse suspicion due to the possibility of the victim itself also requesting this permission during the same launch/session. More specifically, this can happen if an attacker requests a permission that was not granted to the victim (i.e., never requested or denied previously) and the victim also requests this permission during the same launch for its current utility, causing back-to-back permission requests. Please note that the same thing is not possible for when attackers request granted permissions because victim apps will not be able to re-request permissions that were granted to them due to the restrictions on the Android platform (i.e., Permission dialogs for granted permissions will not be shown.). We show with our user study that multiple requests in a single launch within a short time period indeed raise suspicion in users, who then consider investigating the underlying reason and taking action. For example, users might get suspicious of the requesting app and remove it, which causes the attacker to lose a victim app. They can also get suspicious of other apps on their devices and consequently discover and uninstall the attacker, or they can mistakenly put the blame on another victim app and remove it instead. They can also become suspicious of the operating system itself and attempt to reformat their device, by which the attacker faces the possibility of being swiped off the device along with its victims. Since the attacker has no control of when the victim can request permissions, it is safer for them to target granted permissions which they know cannot be requested during the same session to minimize the risks. Additionally, we show with our user study that the likeliness of a user granting a previously-granted permission is statistically similar to that of granting a permission for the first time; hence, the adversary is not compromising effectiveness with this choice. We implement this strategy in our attacks and only request permissions granted to victims.

Previous work has also shown the reputation of an app developer to be another major decision factor for users to grant permissions, consistently for all permissions [2]. For this reason, we recommend that the attacker utilizes victim apps that are highly-popular and have gained users' trust in order for the attacks to be successful. It is worth noting that we have also devised a way for adversaries to expand their attacks to multiple victim apps *simultaneously* by utilizing the lack of app name verification against Google Play (GP) listing names. Such a multi-target attack scheme can be desirable over a single-target scheme when the attacker needs multiple permissions that can only be provided by a combination of victim apps, in line with our idea of our attacks providing an attack platform for adversaries to obtain *any* of their desired permissions. In addition, this scheme gives the adversary more chances to deploy their attacks, as there are now multiple apps that can be targeted. To elaborate, the attacker chooses a name that can *logically* represent any foreground app when displayed in a permissions dialog, mischievously taking advantage of the plain English interpretation of the question displayed in the permission dialog. More specifically, the adversary selects **this app** as their attack app's internal name and the question in the permission dialog will now read as "Allow **this app** to access your contacts?", as shown in Figure 1c. Clearly, this question's plain English meaning does not distinguish between apps and is capable of referring to *any* app that is *currently* in the foreground for a given permission request. We have verified that such an app is accepted to GP and can be installed on a user device without any issue. We have also verified with our user study that the majority of users (199 out of 200) do not seem to notice anything unusual with this particular app name, which we believe is an indication that users are generally not aware of the identity guarantee provided by app names in permission dialogs.

In addition, it is worth mentioning that our attack benefits from certain UI tricks for its successful execution. First, we observed that after a user makes their permission decision, since the adversary is using an invisible activity, the top notification bar of the Android system also appears as a part of the transition effect, creating a suspicious look. In order to ensure the visual effect is subtle enough to not be noticed, we first temporarily hide the top bar at the time of the permission request using the `Window.setFlags()` API with the `FLAG_FULLSCREEN` flag. After the user is done with their decision, the top bar is automatically re-enabled by the system. Second, users can view running apps via the recents screen, which is an avenue of detection or at the very least for getting killed for the attacker. To avoid this, the attack app can hide itself from this screen by setting the `android:excludeFromRecents` flag of its activities in its manifest.

Attack steps. As we have explained the overall idea of our attacks and our methodologies, we will now give a step by step guide to launching false transparency attacks.

1) *Lurk in the background.* The attack app creates a service to continuously run in the background and periodically collects information about the running apps to determine when to attack. Prior to Android 8, running background services could be achieved by using `Service` or `IntentService` classes. However, Android 8 brings restrictions to background execution of apps in order to preserve device resources to improve user experience. Background services are now killed a few minutes after the app moves to the background; hence, the use of `JobScheduler` is more appropriate for our attacks [8] as JobSchedulers can indefinitely run in the background to periodically execute jobs. Additionally, the adversary will also avoid situations that might arouse suspicion in the user. In particular, the app will not use its spoofed name in its launcher name shown in the app menu and instead set it according to the declared legitimate use of the app listed in the respective app store, in order to prevent a possible detection by the user.

2) *Choose your victim carefully.* The attack app runs our ProcHarvester-based implementation to detect victims in the foreground. This entails continuously monitoring the proc filesystem and running our real-time foreground app inference algorithm, which we will describe in more detail in Section 5.

3) *Choose your permission carefully.* Once we obtain the foreground app, we will query the `PackageManager` to obtain the requested permissions of this app and prompt the user for a permission that was granted to the victim but not to the attacker. If there are multiple such permissions, we will randomly pick one to request. Please note that more intricate selection algorithms can be used to more properly pick the permission to be requested. For example, previous work has shown that microphone permission is the most denied permission and hence can be considered the most valuable from the perspective of our attack [2]. In this case, the attacker

might want to prioritize the microphone permission if the foreground app can make a very good case of needing the microphone (e.g., music app or communication app). However, we do not perform this kind of advanced permission selection as our main purpose is to demonstrate our attacks realistically without overly complicating our implementation.

4) *Cloak and dagger.* Once the attacker determines that a certain permission should be requested from the victim in the foreground, they will start an invisible activity from the background service via the `startActivity()` API. This activity will then be automatically moved to the foreground as we have previously explained in Section 4.1. Then, the attacker requests the chosen permission from the context of this invisible activity using the `requestPermissions()` API.

5) *Leave no trace behind.* Once the user completes the permission request, the attacker will call the `moveTaskToBack()` API in order to move to the back of the activity stack to evade detection and continue running silently. This way, the victim app will be restored back to the foreground and the user can continue interacting with the victim.

5 Foreground App Inference

As we have described in Section 4, the adversarial app running in the background will continuously monitor the foreground to detect known victim apps to target with false transparency attacks. Here, we will explain the previous efforts for foreground app inference, why they fail to work in realistic scenarios, and our approach for effectively inferring the foreground app in real time.

Past efforts for foreground app inference. Previously, Android offered convenient APIs, such as `getRunningTasks()`, that could be used to infer the identity of the foreground tasks; however, these APIs have been deprecated in Android 5 in an effort to maintain the privacy of the running apps and prevent phishing attacks on the platform. This has consequently led to a search to identify other avenues that can accomplish the same task. Having inherited many features and security mechanisms from Linux, Android, too, has a proc filesystem (procfs) that provides information about the running processes and general system statistics in an hierarchical structure that resides in memory. Security researchers have discovered that Android's proc filesystem provides numerous opportunities for attackers to infer the foreground app [9, 10]. In response, Android has been gradually closing access to all the sensitive parts of the procfs pointed out by researchers in order to prevent phishing attacks. In the most recent Android versions, all of per-process information on the proc filesystem has been made private (i.e., accessible only by the process itself) and only some of the global system information have been left to be still publicly available due to utility reasons, rendering the efforts to identify the foreground app virtually impossible.

More recently, though, Spreitzer et al. discovered that despite all the strict restrictions on the procfs, there are still public parts of this filesystem that initially seem innocuous but in fact can be utilized to effectively identify the foreground app by employing a relatively more complex analysis in comparison to the previous efforts. To this end, they introduced a tool named ProcHarvester that uses machine learning techniques to automatically identify the procfs information leaks (i.e., foreground app, keyboard gestures, visited websites) on potentially all Android versions, including the newer versions with limited procfs availability, by performing a time series analysis based on dynamic time warping (DTW) [7]. Then, they showed that these identified parts can be utilized for foreground app inference via a similar technique, yielding a high accuracy. In particular, ProcHarvester comprises of two main components: 1) a monitoring app that logs the public parts of procfs on the user device and 2) a server as an analysis unit (connected by wire to the phone) that collects this information from the monitoring app to first build profiles of app starts for the apps in their dataset and then perform DTW to identify information leaks. ProcHarvester currently works as an offline tool in a highly-controlled environment and is not capable of inferring the foreground app in real time, which is an absolute necessity for our attack scenario.

Real-time foreground app inference under realistic scenarios. In our work, we build on ProcHarvester for inferring the foreground app in our attacks. More specifically, we modified ProcHarvester to adapt to realistic scenarios and implemented real-time inference of time series to identify the foreground app. Here, we utilize 20 high-profile apps to serve as the victim apps that the adversary will primarily target for permissions. In addition, we have 380 apps that we will not utilize as victims but use in our experiments to show we can distinguish between victim and non-victim apps at runtime. We chose our victim apps to be from the same dataset as in the original ProcHarvestor work in [7] while we utilized the top apps from each category on Google Play as our non-victim apps. Coverage of permission groups utilized by the apps in our dataset can be observed in Table 2. We deployed our implementation and performed our experiments on a Google Pixel device that runs Android 7.0.

We first ran the original ProcHarvester implementation to create profiles of only the procfs resources that yielded high accuracy for app inference [7], for each victim app in our dataset. Additionally, in original ProcHarvester system, the analysis unit (server) is directly connected to the user device by wire and is collecting data from the device through this connection. However, in our case, adversaries cannot assume a wired connection to a user device as this does not constitute a realistic attack scenario. Hence, we modified the monitoring app to send continuous data to a remote server, which is running our foreground app inference algorithm in

Table 2: Permission distribution for the apps in our dataset.

Permission Group	# of victim	# of non-victim
CALENDAR	1	28
CALL_LOG	0	7
CAMERA	7	207
CONTACTS	14	170
LOCATION	12	228
MICROPHONE	6	95
PHONE	20	376
SENSORS	0	2
SMS	2	9
STORAGE	20	315

real time. This is a plausible assumption as adversaries can easily obtain the install-time `INTERNET` permission, which is of normal protection level, to communicate over the Internet.

Most importantly, we implemented a *real-time* dynamic time warping (DTW) algorithm to detect the foreground app. Currently, ProcHarvester can only be used as an offline inference tool, as it works based on the assumption that app launch times will be known to the tool in advance and the tool can run its DTW-based analysis starting from those launch times. However, this assumption is unrealistic in real-life scenarios as an attacker cannot assume to have a priori knowledge regarding app launch times since an app launch is either at the user's discretion or is initiated by the system or other apps via IPC. In our work, we devise a technique to identify an interval of possible values for the app start time and run DTW starting from all possible values in this interval, rather than using a single starting point as in the original ProcHarvester, to obtain the foreground app *in real time*.

First, in order to obtain the starting time of an app launch, we utilize the `getRunningTasks()` API to monitor foreground changes. Even though this method was previously deprecated as a countermeasure for phishing, we observed that it still provides limited information regarding the foreground of the device. For example, on Android 5-8, whenever there is an app in the foreground, the `getRunningTasks()` API outputs the package name of the caller app (regardless of it being in the foreground or not), and if there is no app in the foreground, it outputs `com.google.android.apps.nexuslauncher`, which corresponds to the Android launcher menu. By continuously monitoring such foreground changes, we can know if an app launch has been *completed* if the foreground state changed from "no app" to "some app", providing us the approximate *end time* (α) for the launch operation. The same information can be obtained on Android 9 with a similar technique as explained in Section 4.2. Now, if we know the duration of an app launch event, we can subtract this from the end time to find the approximate start time of the app launch event. To identify this duration, we run an experiment on our victim dataset and show that app launch takes around $379ms$ on average with a standard deviation of $32.99ms$, which gives us the final range of $[\alpha - 379 - 32.99, \alpha - 379 + 32.99]ms$ for all possible app start times. For each app in our dataset, we then calculate the DTW-based distance using each of the possible values in this interval as the starting point of the analysis and take their average to obtain the final distance. Lowest of these distances corresponds to the foreground app.

Please note that the original ProcHarvester also makes a closed-world assumption: it assumes the app in the foreground that is to be identified is *always* a known, profiled app. This means that the distance reported by ProcHarvester for an unprofiled app by itself does not provide much value in terms of correctly inferring the foreground app since this app's profile is unknown by ProcHarvester. It is imperative for our attacks to be launched only when one of our victim apps is in the foreground. In addition, it is simply impractical to profile all existing Android apps. Hence, we need a mechanism to extend ProcHarvester to distinguish between victim (profiled) and non-victim (unprofiled) apps at any given time. For this purpose, we fingerprint each of our victim apps (app i) by recording the mean (μ_i) and the standard deviation (d_i) for 10 runs where the algorithm correctly identifies app i to be in the foreground. Then, if the lowest calculated distance for a given foreground app is less than or equal to $\mu_i + d_i$ ms to its closest match, we consider this app to be one of our victims.

In order to evaluate our foreground app inference implementation, we conducted experiments where we launched each of the 400 apps in our dataset 10 times and reported the overall accuracy and performance. Our experiments indicate that our algorithm correctly infers the foreground app (i.e., output its identity if it is a victim app or report if it is a non-victim app) 90% of the time. Furthermore, we find the total time to infer the identity of an app in the foreground (after its launch) to be 7.44s on average with a standard deviation of 1.62s. We consider this to be a reasonable delay for our attacks as we expect users to stay engaged with one app before they switch to another for much longer than this duration (18.9s or more on average) [11]. Since the foreground app will presumably not change during the analysis, the adversary should not have a problem targeting the identified app in their attack after this introduced delay. In addition, please note that the original ProcHarvester itself needs around five seconds of procfs data to correctly compute the foreground app.

It is worth mentioning that ProcHarvester is inherently device-dependent since an app can have distinct profiles for a given procfs resource on different mobile devices, which would affect the performance of foreground app inference. Hence, in order to launch a "full-blown attack" that can work on multiple mobile devices, adversaries would have to obtain the procfs profiles of their victim apps on all those devices. Here, adversaries could conveniently adopt a strategy to collect the profiles for only the most commonly-used Android devices in order to quickly cover a satisfactory user base. Note that this extra profile data should not greatly affect the performance or accuracy of the foreground app inference, as an attacker can first identify the type of the device in real time via utilizing existing tools [12] and only use the respective profiles in their analysis, avoiding DTW-based comparisons with profiles belonging to other devices. In our work, we utilize the profiles from only one Android device (Google Pixel) as we primarily intend our attacks to serve as a proof of concept.

6 User Studies: Analyze, Design, Evaluate

Since our attack is a phishing attack at its core, it is important that it is persuasive to users. We speculate that users' comprehension of runtime permissions and their expectations from apps in this context will play a significant role in how users perceive our attack and impact its success. To this end, we performed a survey-based user study to quantify user behavior and used this quantification in order to guide the design of the attack and estimate its chances of success. Our findings suggest that Android users generally have a good understanding of the basics of the runtime permission model but appear confused about its intricate details. In particular, users demonstrate significant lack of appreciation of the critical security guarantees provided by runtime permissions. This leaves a sufficient gap in user understanding to enable an effective attack. In addition to the survey study, we conducted an in-lab user study, which involved fewer users than the survey but provided a more realistic setting based on real devices and common daily tasks performed with popular apps. We provided each participant with an Android device on which we launched our attacks and found that none of the participants detected our attack. We obtained IRB approval from our institution prior to the commencement of our user studies.

6.1 Susceptibility and Design

Our survey has two goals. First, we would like to estimate the susceptibility of users to false transparency attacks. Second, we would like to verify the validity of our conjectures on what makes users suspicious so the design of the phishing attack can reflect the best options for deception. Previous work has shown that permission requests not deploying our attack are likely to be denied by users if the app is not highly-reputable or does not provide any utility that requires the requested permission [2]. We treat this as a baseline control compared to our technique. We refer our readers to Appendix A for a more detailed discussion on this.

Recruitment and incentives. We recruited 200 participants from Amazon Mechanical Turk (mTurk) to complete our online survey. Our inclusion criteria are 1) using Android as a primary device, 2) having at least 100 approved Human Intelligence Tasks (HIT), and 3) having a HIT approval rate of at least 70%. We paid each participant $0.5 for their effort. The median time to complete our survey was 7.08 minutes.

Table 3: Participant demographics

Gender	Participants
Male	125
Female	75

Age	Participants
18 - 23	11
24 - 30	61
31 - 40	67
41 - 50	30
51 or over	31

Education	Participants
Up to high school	19
Some college (1-4 years, no degree)	40
Associate's degree	18
Professional school degree	1
Bachelor's degree	96
Graduate Degree	26

Employment	Participants
Arts & Entertainment	11
Business & Finance	23
Education	9
Engineering	18
Health Care	11
Human Resources	4
Information Technology	37
Management	12
Miscellaneous	17
Religion	1
Retail & Sales	17
Retired	4
Self-Employed	24
Student	2
Unemployed	10

Participant demographics can be observed in Table 3.

Methodology. At the beginning of this survey, we informed our participants that they will be asked questions about their experience with Android permissions; however, to avoid unnecessarily priming them, we do not reveal that we are testing the feasibility of our attacks. We ask questions to assess their knowledge of runtime permissions to understand if there is any underlying vulnerability due to lack of domain knowledge. In addition, we ask questions to verify the design decisions we discussed in Section 4.

Results. We now present our findings from this survey. The percentages we quote below have a ±7% margin of error for a 95% confidence. We will specify the questions we obtained these results from to help our readers easily follow our results. Appendix B presents our survey questions in quiz format.

• *Understanding of the runtime permission model.* We first ask users to self-report their level of familiarity with Android permissions. 8% of the users identify themselves as expert, 41% as knowledgeable, 37% as average, 13% as somewhat familiar, and 1% as not familiar (**Q1**). 71% of the users are aware that Android used to have an install-time permission model (**Q2**). The vast majority of users (91%) have used the new runtime permissions (**Q4**) while almost all of the users (98%) are aware that runtime permission model allows them to review and update their previous permission-related decisions through the Settings app (**Q21**). These results indicate that our participants are generally familiar with the basics of runtime permissions.

In contrast, we observe that users' answers are often wrong when we ask more intricate questions about the inner workings of runtime permissions. An app needs to be in the foreground during a permission request, but less than half (47%) of the users agreed with this, while 25% disagreed and 28% said they did not know (**Q24**). This is worrisome because this fact is central to the contextual security guarantee of the runtime permission model as we explained in Section 4.1. Indeed, as we will show, only *one* of the users who agreed was able to use their understanding in practice to avoid our attack.

When participants were asked whether they thought an app could prompt the user again for a permission that was previously granted to it, 41% agreed, 36% disagreed, and 23% said they did not know (**Q10**). This statement is false. Android does not allow apps to re-prompt users for granted permissions: permission dialogs are never shown again to the users in this case. This misunderstanding can be exploited, as shown with our attacks in Section 4.

We ask further questions to assess users' awareness of the identity security guarantee provided by app names in permission dialogs. First of all, we present them with a storyboard of our attacks where we describe an actual scenario concerning a popular app requesting permissions for its use. We ask them to role-play based on screenshots of the permission requests. For this purpose, we utilized Viber, a popular messaging app with millions of downloads. In particular, we presented our participants with a scenario where they use Viber to text their friends and the app requires contacts permission for providing this utility. Then, they switch to another app briefly and switch back to Viber again where they continue texting. Afterwards, we ask them to grant or deny each permission request. The first time they use Viber, the permission dialog displayed to them is benign and we use the name "Viber" (**Q5**). However, the second time we instead display "this app" as the app name in the permission dialog representing our multi-target attack scenario (**Q13**, **Q14**). We observed that 77% of the participants granted the permission for the benign request (**Q6**) and 74% of them subsequently decided to allow the second (malicious) permission request (**Q15**). Note that this difference falls within our 7% margin of error. For participants who denied the second request, we inquired if they declined due to having noticed our attack. For this purpose, we provide them a text field under the "other" category to write their comments. We had only one user who noticed the odd app name and declined the permission because it looked "fishy".

In another role playing example, we presented an actual scenario where they used Google Maps for navigation and the app prompts for the location permission (**Q17**). We again use "this app" for the attack app's name and ask users to grant or deny the permission (**Q18**). In this case, 89% of the users decided to give the app the permission. We then asked them which app they have given or denied the permission to (**Q19**, **Q20**). 168 (84%) of our participants reported that they granted or denied it to Google Maps, while the rest of them had varying answers: 4 said Google, 4 mentioned a map program, 2 could not remember the app name, 6 mentioned another app (i.e., Viber (5), Yelp (1)), 1 said "this app", 3 said "the app" or "the app I use", 8 said they granted the location permission. The rest (4) wrote somewhat irrelevant text, not showing much understanding of what the question is asking. Note that the participant who said "this app" denied the permission request in this case, but they granted the requested permission to Viber for the malicious request. To sum up, we believe the results from both our Google Maps and Viber examples demonstrate that users are generally unaware of the identity guarantee provided in permission dialogs, as the majority fails to recognize anything suspicious. To our attack's advantage, they seem to be mostly interested in the context they are presented with at the time of the request (i.e., what they are seeing); they either do not pay attention to app names in the requests or simply consider the plain English interpretation of the statement shown in the dialogs.

In conclusion, although users demonstrate familiarity with runtime permissions, we observe that they struggle with the more intricate details of this permission model. They especially show lack of understanding of the security guarantees of runtime permissions, thus leaving avenues for a false transparency attack.

• *Verifying the design decisions for the attacks.* In this part of the study, we verify the validity of our design decisions made in Section 4 regarding the best conditions for the attacks. First, we show that it is indeed suboptimal to request a permission when there is no app in the foreground. Second, we show that requesting a permission multiple times within the same session would indeed alarm the users and lead them to consider taking an action. Hence, we should only request granted permissions. Third, success rate of a secondary malicious request is as likely as a primary benign request. We do not study how the relevance of a permission to an app's utility affects users' decisions, as this relationship was previously demonstrated to be correlated with higher grant rates [2].

First of all, we would like to verify it is indeed not ideal for an attacker to request a permission when there is no app in the foreground. For this purpose, we show a sample screenshot of a popular communication app requesting the contacts permission when there is no visible app in the foreground and ask them if they would grant or deny this request (**Q7**, **Q8**). In this case, 53% of the users select deny, 27% select allow, and 20% express that their decisions would depend on additional factors. For when a similar popular communication app requests the contacts permission while in the foreground (i.e., our aforementioned Viber case), we observe the deny rate to be 23%. We perform Chi-squared test on the deny rate with Yates correction and get the p-value of 1.22×10^{-9}. At the confidence level of 0.05, this indicates that the deny rate without a visible app in the foreground is significantly higher than that when a similar popular communication app is in the foreground.

Next, we would like to verify that users would be alarmed and prone to take an action if an app requested the same permission multiple times within the same launch/session (**Q22**). As we had explained, this case happens only if the attacker requests a permission that was not previously granted. In this case, only 17% of the participants said they would ignore and proceed normally, 43% said they would be suspicious of the requesting app, 23% said they would be suspicious of the other apps installed on their device, 15% said they would be suspicious of the operating system itself, and 2% mentioned they would have other ideas. Participants were able to select multiple options for this question, except for the first option which could be answered only exclusively. We additionally ask the participants who did not say they would ignore the multiple requests what actions they would consider taking (**Q23**). 43% said uninstalling the app that requested the permission, 41% said investigating other apps that request this permission via the Settings app, 11% said reformatting the operating system to go back to factory settings, and 5% mentioned taking other actions. Again for this question, participants were allowed to select multiple options simultaneously.

Additionally, we show that the grant rate for a secondary permission request by an attacker is as successful as a first time request for the same permission by a victim, indicating that the attacker is not compromising the success of their attacks by requesting granted permissions. Looking at the aforementioned Viber case, we observed the grant rate for a primary benign request to be 77% (**Q6**) and 74% for a secondary malicious request (**Q15**). Given our 7% margin of error, we observe no statistical difference between these grant rates. This shows that the request of a previously-granted permission can be as effective as a first time request initiated by the victim, while avoiding unnecessarily alarming users.

6.2 Feasibility of the Attacks

In this part of our user study, we launch our attacks in a realistic setting to evaluate the feasibility of our attacks. More specifically, we are interested in whether the participants would at least suspect they are under attack while performing tasks they might come across in their every day life.

Recruitment. In order to evaluate the feasibility of our attacks, we recruited 20 subjects to participate in our in-lab study on a voluntary basis. We advertised our study via word-of-mouth at the research institution where the study was conducted. Our participant pool consists of undergraduate and graduate students who major in computer science or other engineering fields. Some of our participants even have graduate course level background on security and privacy. Hence, we expect this group to be relatively security-conscious, creating notable difficulty for attackers to successfully execute their attacks. We only recruited participants that have used Android. To avoid priming our participants, we advertised our study's purpose to be a measurement of user expectations in terms of performance for popular Android apps and debriefed them after the completion of our study to disclose our real intent.

Methodology. In our experiments, we utilize three popular Android apps as victims: 1) Google Maps, a navigation app developed by Google, 2) Shazam, an app for song identification developed by Apple, and 3) Messenger, a communication app developed by Facebook. For each app, we assign our participants a simple yet realistic task to complete and ask a question about the task upon completion. First, we ask our participant to launch Google Maps to find the walking route between two predetermined points and tell us the duration of this trip. Then, we ask our participants to launch Shazam to identify the song we are playing during the experiment and tell us the name of the song. Finally, we ask our participants to launch Messenger to send a message to one of our test accounts from the test account set up on the provided phone and tell us what response they got in return.

We have three separate attack apps installed on the device, each targeting only one of the victim apps. The attack apps that target Google Maps and Shazam utilize the same app name as their victims (i.e., Maps and Shazam respectively). The attack app that targets Messenger uses "this app" as its app name in order for us to also test for the feasibility of our multi-targeted attack case. At the end of our experiments, we have an exit survey where we ask the participants about their overall experience with the tasks, i.e., whether they have experienced any slowdown and if they have noticed anything strange or unusual during any of the tasks. We also give them the opportunity to provide us feedback at the end of the survey.

We launch our attacks after the user launches a victim app to complete the given task. In order to correctly infer the identity of the foreground app with certainty, we modified the operating system to change the behavior of `getRunningTasks` API–which was modified by Google in Android 5.0 to not provide this information anymore for privacy reasons as described in Section 4.2–to reflect its old behavior. Please note that such a change is not feasible for an attacker in our threat model and is done solely for the purpose of simplifying our experiments to remove the noise that might be introduced due the use of ProcHarvester. With this approach, we can now focus on assessing how realistic our attacks seem to the users and their potential to be effective to be without having to worry about having correctly inferred the foreground app with ProcHarvester when we launched our attack.

Overall awareness. None of our participants were able to notice our attacks, despite the natural tendency for tech-savviness and security-consciousness among them. It appears that they were mostly preoccupied with completing the tasks and only provided feedback regarding the mundane details of the tasks (e.g., Shazam not identifying the song the first time, late receipt of responses via Messenger etc.). A majority of them (18 out of 20) granted all the permissions when presented with the malicious permission dialogs. One of these participants complained about having to deal with too many permission requests but granted all permissions regardless. From the two participants that did not grant all permissions, one granted the microphone permission to Shazam but denied the location and contacts permissions to Google Maps and Messenger, respectively because they thought neither app needed the requested permissions to perform their tasks (which is indeed true). The other one denied the contacts permission to Messenger because the app did not need it for this task but said they granted the location permission to Google Maps because the app requires it for its main utility so they thought it would still be useful to grant. Even these two particularly security-conscious participants did not seem to catch our attacks. In conclusion, we found *all* of our participants to be vulnerable to our attacks. We believe these findings indicate that false transparency attacks are indeed practical.

7 Defenses and Countermeasures

Phishing attacks have been long dreaded on the Android platform as they are hard to detect and protect against [6, 13]. In a classic phishing attack on mobile platforms, the adversary utilizes existing APIs or side channels to identify the victim in the foreground and immediately launches their own attack app which realistically spoofs victim's components (e.g., UI, name etc.). Hence, they mislead the user to believe they are actually interacting with the victim. Here, we discuss some of the existing defense mechanisms against mobile phishing and why they fall short in the context of false transparency attacks. In addition, we present a serious security vulnerability we discovered in a key security mechanism added in Android 10 with the potential to counteract phishing attacks. We then demonstrate the viability of false transparency attacks on this Android version and onward. Finally, we propose countermeasures that can be implemented on the Android platform and on app stores such as Google Play to practically tackle false transparency attacks.

Provenance-based techniques. As a defense mechanism against UI deception, both [14] and [6] advocated for helping users identify the origin of a UI shown on the screen with a security indicator added to the system bar. Unfortunately, these approaches require invasive modifications to the Android framework, which proved their adoption unpractical.

Blocking side-channels. Android's response to phishing attacks has long revolved around blocking access to certain APIs and public resources that provide a medium to obtain the necessary information (i.e., identity of the foreground app) to successfully carry out such attacks. For example, as we have previously explained, the `getRunningTask()` API and similar APIs that provide information regarding the running apps and services on the device have been deprecated in Android 5. In addition, access to the proc filesystem, which provides a side channel to infer the state of the apps running on the device, has been gradually closed down. However, as we have proven with our attacks, these security measures still fall short and only serve as a band-aid to a deeper problem. We argue that it is infeasible to continue putting effort into identifying and closing down all side channels that provide information about the foreground as some of these channels cannot be made private or deprecated due to utility reasons. For instance, monitoring apps depend on procfs to report app statistics. Hence, a different approach might be necessary to address phishing on Android without compromising utility.

Removing key enablers. Our observation is that the main enabler of phishing on Android is *the ability of apps to start activities in the background to replace foreground apps*. If we can stop background apps from surreptitiously replacing foreground apps, phishing attacks can be conveniently addressed on Android. In fact, we observed that Google implemented a security mechanism that adopts this approach in Android 10. Activity starts from background apps will now be blocked on Android unless the app can justify this action, such as by having a service that is bound by the system or by another visible app [15], or by having recently started an activity (within around 10s). Even though this approach might first appear as an effective countermeasure for phishing, we identified ways to evade it and still start activities in the background without satisfying any of the required conditions checked by the system to allow such an operation. Hence, we were able to verify that our attacks still work on Android 10 and later.

In particular, we discovered that there are two main ways that we can start activities in the background without getting blocked by the system. First, background apps are now subject to time restrictions in terms of how long they can stay in the background while still being able to successfully start activities (i.e., 10s grace period). However, one can periodically start an invisible activity around every 10s and immediately move to the back of the task stack again via the `moveTaskToBack` API to retain the ability to start activities in the background at any point. Second, we have discovered that the `moveTaskToForeground` API is not being held subject to the same restrictions by the Android platform; regardless of how long an app has been in the background, it can always call this API to conveniently move to the foreground. These are both serious design issues that hinder the effectiveness of this security mechanism against phishing attacks.

Upon our correspondence with Google, we have learned that the specific attack with `moveTaskToForeground` has been addressed in a more recent revision of Android 10. However, the periodic restart issue created by the 10s grace period seems inherently harder to address and is likely to stay as it might require redesign of the implemented security mechanism. In fact, we verified that the 10s grace period still exists in Android 11, which is available in beta version at the time of writing. In order to at least minimize the practicality of this specific attack vector, Google has attempted at a countermeasure by implementing a mechanism to cancel the grace period on certain user interaction (i.e., pressing the home button). However, we observed that this implementation was also problematic and unfortunately not effective on Android 10. What we have shown here is that addressing the problem is not trivial and guaranteeing correctness may require many versions, redesigns, and steps of testing. In the end, these vulnerabilities have the potential to make our attack more likely to succeed because users will not be expecting activity starts or permission requests from background apps at all on Android 10. Google has acknowledged our findings as a serious security vulnerability that required swift remediation. In addition, this vulnerability was featured in the upcoming Android Security bulletin due to its significance.

Our suggestions. We propose multiple strategies that can be implemented simultaneously or as stand-alone techniques to address false transparency attacks in a practical manner.

• *New app store policies.* False transparency attacks can be addressed at the app store level with the addition of new policies into these stores. For example, Google Play (GP) can implement name checks to ensure the uniqueness of app names across all the apps served on GP. In addition, GP can perform additional checks to catch confusing app names like "this app". Such checks would have to be implemented on all existing app stores to provide uniform security across app markets. However, one can argue that implementing the checks on GP can be sufficient as the majority of trustworthy apps that can be utilized as victims in our attacks are only served on GP. Nevertheless, side-loaded apps will not be subject to these checks performed on app stores.

• *Enforcing app name integrity in the Android framework.* Perhaps a more effective and efficient way of addressing our attacks can be achieved by enforcing the uniqueness of an app name on the Android platform itself. This enforcement can be performed during installation to filter out apps with suspicious app names on a first-come-first-serve basis.

• *Additional app identifiers in the permission dialog.* Currently the permission dialog on Android only contains the name of the app in the dialog to help users identify the app. Additional identifiers, such as an app logo, can be added to the system dialog to remove any confusion regarding the origin of an app. Google Play readily implements mechanisms to prevent logo-based phishing to ensure logos of different apps will not be dangerously similar. Hence, this can indeed be a viable approach in addressing false transparency attacks.

• *Mandatory app transition effects.* In false transparency attacks, one of the problems is that the context change between apps is not visible to the user. In order to make the context change more visible, mandatory transition effects can be added between foreground app switches. This way, when the attacker launches their attack, the user might be able to catch that the request is not coming from the victim app as they have just observed the foreground change. It is worth mentioning that Android 10 attempts to solve this problem by introducing a security mechanism that prohibits apps from starting activities from the background; however, there seems to be design issues with this mechanism as we have explained.

• *Prohibition of transparent activities.* Android platform can ban the use of transparent activities altogether to eliminate phishing attacks that make use of such UI components. Although transparent activities might have some legitimate use cases, we expect these to be limited.

8 Related Work

Mobile UI spoofing attacks. In mobile UI spoofing attacks, users are tricked into misidentifying apps. As a result they inadvertently either provide sensitive information or perform critical operations that will be beneficial to adversaries [6]. These attacks can be classified into two categories. In *phishing attacks*, the adversary surreptitiously replaces or mimics the UI of the victim to lead the user into falsely believing that they are interacting with the victim [13, 16, 17]. Phishing attacks rely on existing APIs or side-channels to identify the foreground app [7, 9, 10, 18]. In *clickjacking*, also known as the *UI redress attacks*, the adversary places opaque and click-through overlays covering either parts or the entirety of the victim. While the user assumes they are interacting with the UI provided by the overlays, their clicks in fact reach the victim where they induce a state change [19–22].

Android permissions. Android's permission model has been subject to much criticism due to a range of issues including its coarse granularity [23], side-channels to permission-protected resources [24–27], and design issues with custom permissions [28, 29]. Previous work has also investigated the effectiveness of install-time permissions and concluded that users would benefit from having the ability to revoke permissions [3]. Micinski et al. conducted a user study on Android runtime permissions and concluded that authorization might not be required for permissions tied to user interactions as users are generally aware that such interactions will result in the utilization of certain permissions [30]. In addition, Alepis et al. discovered transformation attacks on runtime permissions [31], which are similar to our attacks in essence but lack the important execution details (e.g., the design strategies and their implementation, design of multi-targeted attack scenario that expands the attack surface, user studies to support stealthy attacks etc.) that are crucial for the success of the attacks. Finally, [32] presents a preliminary version of our work.

9 Limitations and Future Work

We verified that the vulnerability we describe in this paper (i.e., permission requests with confusing app names from transparent background apps) exists on all Android versions that support runtime permissions (Android 6-11). However, we demonstrated the effectiveness of our foreground app inference technique only on one such Android version (i.e., Android 7) to show the viability of our attacks. The reason is that we expect to obtain similar accuracy results since we found the same procfs resources that we utilized in our analysis to be available and given the significant structural similarities across all these Android versions. In fact, based on the ProcHarvester measurements [7], we estimated that there would be no more than about 5-10% variation in accuracy between the versions. However, we acknowledge that there is still value in performing further experiments on other versions for a more complete analysis and leave this for future work.

Our foreground app inference implementation might have impact on the device's battery life as it currently runs in the background to periodically check for changes in the foreground. However, an adversary can poll for these changes less often by sacrificing some of the attack opportunities. In addition, it seems possible to optimize the periodicity for polling based on how often users change between apps (18.9s or more on average) [11]. We leave the utilization of such techniques for future work. It is also worth mentioning that the methods we use in this work are meant to be modular. If a better approach to foreground inference is developed in the future, an attacker can use that instead.

Finally, our in-lab user study demonstrates the feasibility of our attacks for an ideal condition where the attacker is *always* able to correctly infer the foreground app due to our use of the modified Android version, as explained in Section 6. However, future work is needed to show the feasibility of the attacks under a more realistic scenario where there may be some errors in the foreground inferences made by our ProcHarvester-based technique.

10 Conclusion

In this work, we presented *false tranparency attacks*, a class of phishing-based privilege escalation attacks on Android runtime permissions. We conducted a user study to understand if users' understanding of runtime permissions would innately create susceptibility to these attacks. We designed these attacks to launch strategically in order to minimize the possibility of alerting the user while retaining effectiveness and verified the validity of our design decisions through our user study. In addition, we conducted a lab study to demonstrate the feasibility of our attacks in a realistic setting and showed that none of the participants were able to notice our attacks. We discussed why existing defenses fall short in the context of false transparency attacks. In particular, we disclosed the vulnerabilities in a key security mechanism implemented in Android 10, which consequently allowed us to still launch our attacks on this recent Android version. Finally, we proposed a list of countermeasures to practically defend against false transparency attacks.

Acknowledgments. This work was supported in part by NSF CNS 13-30491 (THaW). The views expressed are those of the authors only.

References

[1] Android app permissions best practices. https://developer.android.com/training/permissions/usage-notes.

[2] B. Bonné, S. T. Peddinti, I. Bilogrevic, and N. Taft. Exploring decision making with android's runtime permission dialogs using in-context surveys. In *SOUPS*, 2017.

[3] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov. Android permissions remystified: A field study on contextual integrity. In *USENIX Security*, 2015.

[4] Android permissions. https://tinyurl.com/y863owbb.

[5] Android dashboard. https://tinyurl.com/qfquw3s.

[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 *IEEE Security and Privacy*, 2015.

[7] R. Spreitzer, F. Kirchengast, D. Gruss, and S. Mangard. Procharvester: Fully automated analysis of procfs side-channel leaks on android. In *Asia CCS*, 2018.

[8] Background execution limits. https://developer.android.com/about/versions/oreo/background.

[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 *USENIX Security*, 2014.

[10] W. Diao, X. Liu, Zhou Li, and K. Zhang. No pardon for the interruption: New inference attacks on android through interrupt timing analysis. In *IEEE Security and Privacy*, 2016.

[11] L. Leiva, M. Böhmer, S. Gehring, and A. Krüger. Back to the app: the costs of mobile application interruptions. In *Human-computer interaction with mobile devices and services*, 2012.

[12] Android device names. https://github.com/jaredrummler/AndroidDeviceNames.

[13] S. Aonzo, A. Merlo, G. Tavella, and Y. Fratantonio. Phishing attacks on modern android. In *CCS*, 2018.

[14] 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 *Financial Cryptography and Data Security*, 2016.

[15] Android Q privacy change: Restrictions to background activity starts. https://developer.android.com/preview/privacy/background-activity-starts#display-notification-user.

[16] A. P. Felt and D. Wagner. *Phishing on mobile devices.* 2011.

[17] E. Chin, A. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *MobiSys*, 2011.

[18] R. Spreitzer, G. Palfinger, and S. Mangard. Scandroid: Automated side-channel analysis of android apis. In *Security & Privacy in Wireless and Mobile Networks*, 2018.

[19] M. Niemietz and J. Schwenk. Ui redressing attacks on android devices. *Black Hat Abu Dhabi*, 2012.

[20] L. Wu, B. Brandt, X. Du, and B. Ji. Analysis of clickjacking attacks and an effective defense scheme for android devices. In *CNS*, 2016.

[21] 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 *IEEE Security and Privacy*, 2017.

[22] A. Possemato, A. Lanzi, S. P. H. Chung, W. Lee, and Y. Fratantonio. Clickshield: Are you hiding something? towards eradicating clickjacking on android. In *CCS*, 2018.

[23] J. Jeon, K. K. Micinski, J. A Vaughan, A. Fogel, N. Reddy, J. S Foster, and T. Millstein. Dr. android and mr. hide: fine-grained permissions in android applications. In *SPSM*, 2012.

[24] D. He, M. Naveed, C. A. Gunter, and K. Nahrstedt. Security concerns in android mhealth apps. In *AMIA*, 2014.

[25] P. Sapiezynski, A. Stopczynski, R. Gatej, and S. Lehmann. Tracking human mobility using wifi signals. 2015.

[26] S. Narain, T. D. Vo-Huu, K. Block, and G. Noubir. Inferring user routes and locations using zero-permission mobile sensors. In *IEEE Security and Privacy*, 2016.

[27] Y. Michalevsky, A. Schulman, G. A. Veerapandian, D. Boneh, and Gabi Nakibly. Powerspy: Location tracking using mobile device power analysis. In *USENIX Security*, 2015.

[28] J. Sellwood and J. Crampton. Sleeping android: The danger of dormant permissions. In *SPSM*, 2013.

[29] G. S. Tuncay, S. Demetriou, K. Ganju, and Carl A. Gunter. Resolving the predicament of android custom permissions. In *NDSS*, 2018.

[30] K. Micinski, R. Votipka, D.and Stevens, N. Kofinas, M. L. Mazurek, and J. S. Foster. User interactions and permission use on android. In *CHI*, 2017.

[31] E. Alepis and C. Patsakis. Unravelling security issues of runtime permissions in android. *Hardware and Systems Security*, 2019.

[32] G. S. Tuncay. *Practical least privilege for cross-origin interactions on mobile operating systems*. PhD thesis, University of Illinois at Urbana-Champaign, 2019.

[33] False transparency attacks. https://sites.google.com/view/false-transparency-attacks/home.

Appendices

A Frequently Asked Questions

Here we discuss some of the concerns that we thought might be raised by the reader, in a Q&A format.

• *Why doesn't the attacker just launch a normal phishing attack?* Google Play (GP) already has some security mechanisms in place to detect phishing attacks. For example, GP does not allow apps to be published with icons that are similar to those of other apps. In addition, GP can also identify if the title of an app is dangerously similar to that of another app. If an app has some suspicious behavior as in these cases, it will be suspended by GP indefinitely. With our attack, the adversary does not have the risk of detection by GP.

In addition, users are generally familiar with the concept of classic phishing attacks, where the attacker impersonates another app by mimicking its UI. Hence, it is more likely, compared to our attacks, that they will be on the lookout for such attacks. Our attacks do not require mimicking another app's UI and are previously unknown to the users (as well as to the research and developer communities). Therefore, users will be vulnerable and will get caught off-guard as they are not expecting such attacks in the first place. We have proven the validity of this statement with our user study.

• *Couldn't the app just pretend to do something useful with the permission to convince the users to grant it?* While it is true that the most important reason for users to grant permissions is the permission's relevance to utility, it is not the sole factor that plays a role in these decisions. Previous work has shown that users consider the relevance of the permission to the app's utility and the reputation of the app developer as a factor influencing them to grant permissions, 68% and 32% of the time, respectively while the average denial rate is reported to be 14% [2]. This means that for an attacker who cannot really make a convincing argument for needing a permission (e.g., QR app needing contact list), they will not be able to obtain the permission 68% of the time with a direct attack. Similarly, for an attacker whose app has not earned much reputation, they will get their permission requests denied 32% of the time. We perform two Chi-squared tests with Yates correction to compare these two denial rates to the average denial rate. The p-values for both tests are much less than 1×10^{-5}, which are much smaller than the confidence level of 0.05. This indicates that a permission's relevance to utility and the requesting app's reputation are both factors that significantly contributes to users' grant decisions.

In general, both our user study and previous studies show that users do try to make conscious decisions when it comes to permissions. They feel more comfortable granting permissions to some apps while they do not feel so for others based on several factors. Our goal is to enable adversaries to take advantage of the user's trust in another app, without having to gain that trust on their own.

• *What is the attacker going to do with the permissions?* Our attacks serve as a platform for different adversaries to obtain the permissions they need to realize their goals. Each adversary can come up with a different attack strategy that requires them to obtain a specific set of permissions, which they could achieve using our attack scheme.

• *Why does Android allow invisible activities?* Android, being the liberal operating system it is, aims to provide app developers the UI design freedom they need to achieve their purposes. Transparent UI features often improve user experience. On the other hand, restrictions to transparency are hard to implement. Android provides a range of transparency options, and it is unclear which ones can be utilized for attack scenarios similar to those we have illustrated in this paper. Indeed, a modification to restrict transparency could complicate the code in a way that introduces new types of vulnerabilities.

B Survey Questions

Here, we present the questions that we used in our survey, results of which we elaborated in detail in Section 6. Statements regarding the display logic (e.g., skip to question X etc.) are not displayed to the participants and are included only to provide our readers an accurate view of our survey. We shuffle answer options at the time of participation in the survey. We do not present our attention check and demographics questions here for brevity. In addition, we removed some questions that we did not utilize in Section 6 for the sake of brevity, but the missing questions have the potential to prime the users to be security conscious, making them more likely to deny permissions and think of defenses. Curious readers can find our full set of questions in [33].

Q1: Please choose your level of knowledge of Android permissions.

- Expert
- Knowledgeable
- Average
- Some familiarity
- No familiarity

Q2: On older Android versions, permissions required by an app were displayed at the time of installation and the installation would not proceed if you did not agree to grant all the listed permissions to the app.

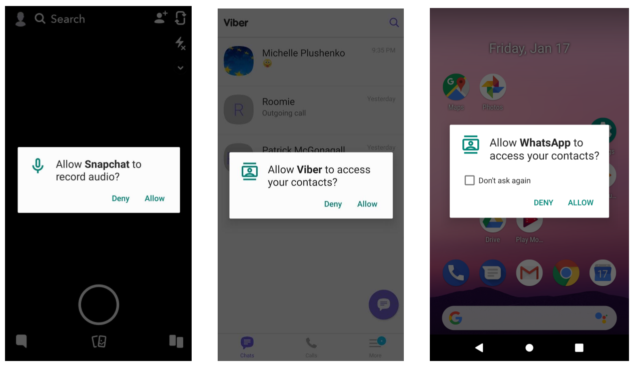
- True
- False
- I don't know

Q3: For your information, on more recent versions of Android, apps can prompt you at runtime with a permission request (also called permission prompt) to get access to some of the device resources. Such permissions that are requested when the app is in the foreground are called runtime permissions. The following screen is an example of a permissions request. You can give access to the app for the resource in question by choosing "Allow". Similarly, you can deny access by choosing "Deny". Please note that how this prompt screen looks might slightly vary depending on the device and Android version.

Q4: I remember seeing a similar permission screen while using my device (It could be for a different app and/or permission).

- Yes
- No
- I am not sure

Q5: Let's do some role-playing now. Suppose for the sake of this survey that you have installed Viber, a popular messaging app with millions of downloads that allows you to communicate with your friends. Viber requires access to your contacts to allow you to contact your friends. If you



(a) Q3 screen (b) Q5 screen (c) Q7 screen

Figure 2: Screens for Q3, Q5, and Q7

don't grant this permission, the functionality likely will not work. Suppose you have started Viber to message a friend and Viber prompted you to get permission to access your contacts for the first time, as shown in the following screen.

Q6: Please decide if you would like to allow Viber to access your contacts.

- Allow
- Deny

Q7: Continuing with the role-playing... Suppose now that you received a permission request from an app installed on your phone while you were not actively using any app. Below screenshot is an example of such a request.
Q8: What would you do about such a permission request?

- Allow
- Deny
- It depends

Q9: An app can request a permission and prompt you again in the future even if you might have denied this permission previously.

- True
- False
- I don't know

Q10: An app can request a permission and prompt you again in the future even if you might have granted this permission previously.

- True
- False
- I don't know

Q11: Suppose you previously denied a permission to an app you currently have on your device. If you were prompted again for the same permission by this app in the future, would you grant it?

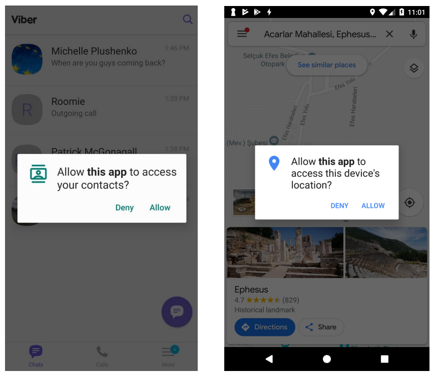
- Yes - Skip to

- No - Skip to Q13
- It depends - Skip to Q12

Q12: Which of the following conditions would influence you to grant the permission after denying it previously? (Choose all that apply)

- The requested permission is necessary for the app to work
- The request is for a permission I do not care about or do not consider particularly risky
- The requesting app is highly popular (i.e., installed by millions of users)
- The requesting app is developed by a well-known company
- Other (Please specify): __________________

Q13: (Display if the first option of Q6 is chosen) Now back to role-playing again. Suppose you texted a couple of friends on Viber, then you switched to some other applications or perhaps stopped using your phone for a while. Eventually, you switched back to Viber to continue texting your friends and now you are prompted for the contacts permission as shown in the following screenshot.
Q14: (Display if the second option of Q6 is chosen) Now back to role-playing again. After being on your phone for a while and doing useful things, you switched back to Viber to text your friends and now you are prompted for the contacts permission as shown in the following screenshot.



(a) Q13, Q14 screen (b) Q17 screen

Figure 3: Screens for Q13, Q14, and Q17

Q15: What would you do?

- Allow - *Skip to Q17*
- Deny - *Skip to Q16*

Q16: What is the reason for denying the permission?

- I already granted this permission to Viber so it should not ask me again
- I already denied this permission to Viber so it should not ask me again
- I always decline permissions
- Multiple requests for the same permission made me suspicious of Viber
- Other (Please specify): __________________

Q17: Suppose you are traveling the world and you found yourself wanting to go to the magical ancient Greek city of Ephesus. You open Google Maps to navigate to these ruins. You are prompted with a permission dialog as in the following picture.
Q18: Which option would you select?

- Allow - *Skip to Q19*
- Deny - *Skip to Q20*

Q19: Just asking to make sure we are on the same page... Which app did you just grant the location permission to? - Skip to Q21

____

Q20: Just asking to make sure we are on the same page... Which app did you just deny the location permission to?

____

Q21: On Android versions that support runtime permissions, you are allowed to grant or revoke permissions to apps at any time by modifying permission settings via the Settings app.

- True
- False
- I don't know

Q22: What would you think if an app has requested a permission it had previously requested during the same launch (i.e., after you started the app it requested the same permission twice within a small time frame)? Please select all that apply.

- I would not think anything of it and proceed with granting/denying the permission normally.
- I would be suspicious of the requesting app.
- I would be suspicious of the other apps I have installed that use this permission.
- I would be suspicious of the Android operating system itself.
- Other (Please specify): __________________

Q23: (Display if the first option of Q22 is not chosen) What would you consider doing in this case (i.e., when an app requests the same permission twice during the same launch)? Please check all that apply.

- Uninstalling the app that requested the permission
- Investigating other apps that request this permission via the Settings app
- Reformatting the operating system to go back to factory settings
- Other (Please specify): __________________

Q24: An app has to be in the foreground (i.e., showing on the screen) when it prompts you for a permission.

- True
- False
- I don't know