Front.Comput.Sci.
DOI

RESEARCH ARTICLE

VenomAttack: Automated and Adaptive Activity Hijacking in Android

Pu Sun[1#], Sen Chen[2#], Lingling Fan[3], Pengfei Gao[1], Fu Song(✉)[1], Min Yang[4]

[1]School of Information Science and Technology, ShanghaiTech University, Shanghai, 201210, China
[2]College of Intelligence and Computing, Tianjin University, Tianjin, 300350, China
[3]College of Cyber Science, Nankai University, Tianjin, 300350, China
[4]School of Computer Science, Fudan University, Shanghai, 200438, China
[#]Co-first Author

© Higher Education Press 2021

Abstract Activity hijacking is one of the most powerful attacks in Android. Though promising, all the prior activity hijacking attacks suffer from some limitations and have limited attack capabilities. They no longer pose security threats in recent Android due to the presence of effective defense mechanisms. In this work, we propose the first automated and adaptive activity hijacking attack, named VenomAttack, enabling a spectrum of customized attacks (e.g., phishing, spoofing, and DoS) on a large scale in recent Android, even the state-of-the-art defense mechanisms are deployed. Specifically, we propose to use hotpatch techniques to identify vulnerable devices and update attack payload without re-installation and re-distribution, hence bypassing offline detection. We present a newly-discovered flaw in Android and a bug in derivatives of Android, each of which allows us to check if a target app is running in the background or not, by which we can determine the right attack timing via a designed transparent activity. We also propose an automated fake activity generation approach, allowing large-scale attacks. Requiring only the common permission INTERNET, we can hijack activities at the right timing without destroying the GUI integrity of the foreground app. We conduct proof-of-concept attacks, showing that VenomAttack poses severe security risks on recent Android versions. The user study demonstrates the effectiveness of VenomAttack in real-world scenarios, achieving a high success rate (95%) without users' awareness. That would call more attention to the stakeholders like Google.

Keywords Android; Activity hijacking; Android security; Mobile security

1 Introduction

Various hijacking attacks in Android have been proposed, such as component hijacking [1], clicking hijacking [2] activity hijacking [3, 4], and task hijacking [5, 6], where task hijacking also can achieve activity hijacking. Among them, activity hijacking which injects into the foreground a hijacking activity, is one of the most powerful attacks. It can breach the integrity or availability of the GUIs belonging to other apps without user's awareness, causing severe consequences in practice.

Though promising, all the existing activity hijacking attacks [3–9] suffer from one or more of the following limitations. (1) It is assumed that a malware designed to hijack a chosen app has been installed on the victim's device, which prevents the adversary from adaptively choosing target apps and devices. (2) The problems of how to identify vulnerable devices to attack are not considered. (3) The malware is created manually for each chosen target app which works for determined attacks but is not suitable for adaptive and large-scale attacks. (4) Almost all the attacks either do not consider the right attack timing or the way to determine the right attack timing have been deprecated or restricted in recent Android versions.(5) Prior attacks become ineffective due to effective

Received March, 2021; accepted July, 2021

E-mail: songfu@shanghaitech.edu.cn

defense mechanisms [6, 8, 10–14] which detect attacks before app installation or at runtime. Thus, it is fair to say that prior activity hijacking attacks no longer pose an effective and powerful security threats in recent Android versions.

In this work, our goal is to investigate if it is possible to overcome the above limitations via more advanced activity hijacking attacks. Thus, the research in this article is motivated by the following questions: (Q1) How to identify devices which installed vulnerable apps and how to install malware on the chosen vulnerable devices, as not all the devices are vulnerable and not all the vulnerable devices are worth to attack. (Q2) How to determine the right attack timing, as it is suspicious to abruptly inject a malicious activity into the foreground. (Q3) Is it possible to mount automated large-scale attacks in recent Android versions, by which the adversary can gain more benefits with low attack cost. (Q4) Is it feasible to bypass the state-of-the-art defense mechanisms that were proposed to defeat activity hijacking attacks?

To address Q1, we propose to leverage hotpatch techniques. Hotpatch techniques were proposed to fix bugs and add new functionalities for installed apps, without re-installation and re-distribution. An app with a hotpatch framework but without any attack payloads could be distributed via app markets (e.g., Google Play Store). It can be customized with useful and interesting functionalities to attract users to install. When the app is installed, it can collect information of the device and installed apps, and send back to the adversary. This allows the adversary to identify and choose vulnerable devices to attack. Then attack payload can be created at the server-side and updated into the app via the hotpatch framework, without re-installation and re-distribution. Furthermore, the attack payload could be removed later so that the app looks like benign after a successful attack.

To address Q2, we present one new flaw in Android and one new bug in derivatives of Android, each of which can be used to determine a right attack timing. The flaw was originally designed to check whether an activity is the first one or not in its task. The bug comes from a flag that indicates if an app is in the force stopped state or not for controlling broadcasts. We found that each of them can be used to infer whether the target app is running in the background or not, allowing us to determine the right attack timing.

We address Q3 by presenting an automated fake activity generation approach for phishing attacks via activity hijacking, one of the most important attack examples of activity hijacking [3–6]. It is challenge to automatically and efficiently generate fake activities as the source code of target apps/activities are often unavailable. We propose an approach to automate fake activity generation. For each target app, it creates a fake activity by launching the target app, taking a screenshot of the login UI, extracting layout information from the login UI, creating a layout file as the fake UI and adding source code to implement the functionality. This leads to an automated, adaptive and large-scale activity hijacking attack.

For Q4, we first argue that an app with a hotpatch framework is benign, thus, all the defense mechanisms that detect activity hijacking attacks before app installation can be bypassed. To justify this, we implemented a bait app with a hotpatch framework, called EasyNote. It was labeled as benign by all the 63 security engines in VirusTotal and was also successfully submitted to Google Play Store by us. Furthermore, we studied 250 popular apps from 5 app markets, out of which 108 apps use hotpatch frameworks, indicating that hotpatch cannot be used as a key indicator for identifying our attack. Remark that activity hijacking apps without using hotpatch techniques can be detected by existing tools [6, 8, 10, 12, 13]. We also design an elegant transparent activity in our attack in order to insert a malicious activity into the foreground at the right timing. It neither starts activities in the background which is restricted in Android 10 nor destroys the GUI integrity of the foreground app which could be intercepted by real-time defense mechanisms, e.g., WindowGuard [11].

To demonstrate the effectiveness of VenomAttack, we first evaluate its compatibility on 6 recent official and 5 commercial Android versions, where the cumulative adoption of the official versions is 82.91% reported by StatCounter on Feb. 2021. We show they are all vulnerable. We then evaluate the efficacy of our fake activity generation by comparing the similarity between fake and original UIs of 50 popular financial and social apps from Google Play Store. The similarity is close to 100% and the generation time is close to 3 seconds. We also thoroughly investigate VenomAttack under the state-of-the-art defense mechanisms that were proposed to defend against activity hijacking attacks. VenomAttack is able to defeat all these mechanisms in principle, while no prior activity hijacking attacks can. We further present proof-of-concept (PoC) attack examples to demonstrate VenomAttack. Finally, we conduct a user study using EasyNote and 10 randomly selected target apps from the top 100 financial apps on Google Play Store. It achieves 95% attack success rate and none of the 20 participants perceive any abnormalities when conducting attacks.

In summary, our main contributions are:

- We present a novel attack, VenomAttack. It is more practical and powerful than prior activity hijacking attacks.
- We propose to use hotpatch techniques and present a new flaw and a new bug, which would call more attentions to the new security issues for different stakeholders.
- We conduct extensive experiments and user study, which demonstrate the effectiveness of VenomAttack which successfully obtains the credentials from almost all the victims in practice.
- We conduct a systematic study of existing activity hijacking attacks under various defense mechanisms, revealing their advantages and disadvantages which potentially fosters further research.

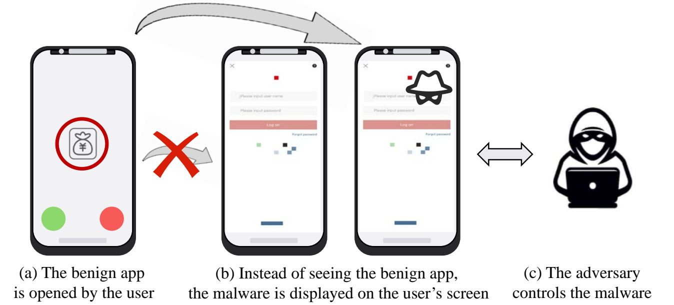


Fig. 1 Attack scheme of activity hijacking

2 Background

2.1 Android Application, Activity, and Task

An Android app consists of one or more components, each of which is *activity*, *service*, *content provider*, or *broadcast receiver*. Basically, an app runs in its own limited-access sandbox and communicates with others via intents. Android enforces a permission-based security policy, where an app can use resources outside of its own sandbox when the declared permissions are granted. While an activity of an app provides a window in which the app draws its GUI, served as the entry point for user interaction. An app can have several activities for different interactions [15].

Android provides a multitasking mechanism to organize *running activities* (i.e., activities that are launched, yet have not been destroyed). A task, called *back stack*, is a collection of running activities. Android arranges multiple tasks as a task stack: the foreground task that interacts with the user, and the background tasks that organize paused activities. When a task comes to the foreground (e.g., by beginning a new task or resuming an existing one), the top activity of the foreground task is displayed on the screen. When the foreground task finishes, it pops from the task stack, the Home screen if the task stack becomes empty otherwise the recent task, comes to the foreground. When an activity starts, there are three basic attributes to determine the resulting task stack: *launch mode*, *task affinity*, and *intent flag*. For instance, when the current activity creates a new activity, the current activity is stopped but remains intact in the task, and the new activity is pushed onto the foreground task. If an activity is pushed onto a newly created task stack, it becomes the root of the task stack which the task affinity is the task affinity of the activity. If the Back button is pressed, the current activity is popped from the foreground task and the next activity in the foreground task resumes. Details and formal definitions refer to [6, 16].

2.2 Hotpatch in Android

Hotpatch is a technique for repairing bugs and adding new functionalities of an installed app without re-distribution and re-installation. Several hotpatch frameworks such as Tinker, Sophix, Robust, and AndFix, varying in the underlying techniques, have been developed. Among them, Tinker, Sophix, and Robust are compatible with the recent Android versions. Typically, to update a base app with a client-side hotpatch framework, a patch file is created by the server-side hotpatch framework from the base app and the updated app, and then is sent to the base app installed on the devices. The client-side hotpatch framework enforces that the updated app will be started when the app is launched. All these steps can be done automatically and stealthily.

In this work, we choose Tinker to demonstrate our attack. Tinker uses the Android's dex loading and Java reflection mechanisms to achieve hotpatch. It supports the update of methods, classes, .so library files and resource files without requiring any additional permissions except for the INTERNET permission. We checked and confirmed that Robust can be used for our attack as well, while Tinker supports more features and is much easier to use. Moreover, the patch file generated by Tinker is much smaller, hence causes less traffic than that of Robust. Note that we did not check Sophix and AndFix, as Sophix is not publicly available while AndFix has not been maintained for several years and replaced by Sophix.

2.3 Activity Hijacking in Android

Figure 1 shows the overview of activity hijacking in Android [3–5]. The goal is to hijack the user interaction of a benign activity by injecting a malicious one. When a benign app or its activity is launched or resumed by the user, instead of seeing the GUI of the benign app/activity, the GUI of a malicious activity is displayed on the user's screen so that the user will interact with the malicious one without user's awareness. Activity hijacking can implement various attacks such as phishing, spoofing and DoS attacks.

3 Related Work

Hijacking attacks in Android have been extensively studied such as component hijacking [1], clicking hijacking [2] activity hijacking [3, 4] and task hijacking [5, 6]. Component hijacking exploits vulnerable components in target apps to carry out unauthorized read or write operations on sensitive resources. Clicking hijacking tricks victims to click on the elements in a different UI page that is only barely visible or completely invisible. Intent hijacking aims to manipulate or steal broadcast intents or certain activity-related intents. This work focuses on activity hijacking, one of the most powerful hijacking attacks, which can implement various attacks (e.g., phishing, spoofing, and DoS) by requiring only the common permission INTERNET. We discuss existing activity hijacking attacks below.

The first activity hijacking attack was proposed in 2013 [7]. It uses a background service to constantly check if the target app is running via the API function `getRunningAppProcesses`. It launches a malicious activity and injects it into the foreground if the target app is running.

In 2014, an alternative activity hijacking attack was proposed [3]. This attack detects the event of the target activity landing by leveraging side channel information (e.g., shared-memory) so that hijacking occurs at the right timing. However, this attack requires a carefully designed timing so that the malicious activity will not enter the foreground too early or too late, and the detection of the target activity landing event is less reliable than API functions [5]. A similar idea was also used Yang *et al.* [9] to launch activity hijacking attacks.

In 2015, a spectrum of task hijacking attacks was proposed [5], which can achieve activity hijacking. They presented two attack types: *malware⇒target* where the malicious activity gets pushed onto the target app's task, and *target⇒malware* where the activity of the target app is tricked by the malware and pushed onto the malware's task. Note that the attack of [7] belongs to *malware⇒target*. To be less suspicious, *target⇒malware* has to detect the target activity landing event, while *malware⇒target* has to detect if the target app is running. It was shown in 2019 that task hijacking attacks exist in real-world malware (36 malcious apps) and all of the top 500 most popular apps are at risk.

In 2016, a *malware⇒target* attack was proposed [4] which uses the API function `getRunningTasks` to check if the target app is running. Though the API function `getRunningTasks` has been restricted later for direct usage, it still works in 2017 if the malware and the target app are in the same task [8]. All possible combinations of the task related attributes are identified for *malware⇒target* attacks by exploring the formal semantics of Android's activity activation mechanism [6]. Recenty, StrandHogg 2.0 was proposed which is executed through Java reflection, allowing the malware to freely assume the identity of target apps.

As mentioned in Section 1, prior activity hijacking attacks suffer from one or more limitations, hence no longer pose an effective and powerful security threats in recent Android versions. Our attack is designed to overcome these limitations. It also features both *malware⇒target* and *target⇒malware* attacks, and enables a spectrum of customized, large-scale attacks without user's awareness.

Fortunately, various defense mechanisms have been proposed to defend against activity hijacking attacks. Static analysis and/or dynamic analysis methods [6, 8, 10, 12, 13] can check if an app contains any activity hijacking attacks that leverage task related attributes. All the API functions used by the existing attacks have been effectively restricted in recent versions of Android. Real-time detection methods also have been proposed [11, 14, 17] to intercept activity hijacking attacks at runtime. An in-depth analysis of prior activity hijacking attacks against these defense mechanisms is given in Section 5.

Our attack is also closely related to piggybacked apps and direct inter-app code invocation (DICI). Piggybacked apps, obtained by repackaging malicious payloads with popular (benign) apps, provide another way to spread to a large user base [18]. Piggybacked apps contain malicious payloads when distributed via app markets, whereas our attack contains a client hotpatch framework instead of a malicious payload. Inspired from piggybacked apps, the adversary could also piggyback a client hotpatch framework into some popular (benign) apps instead of using our bait app and then up-

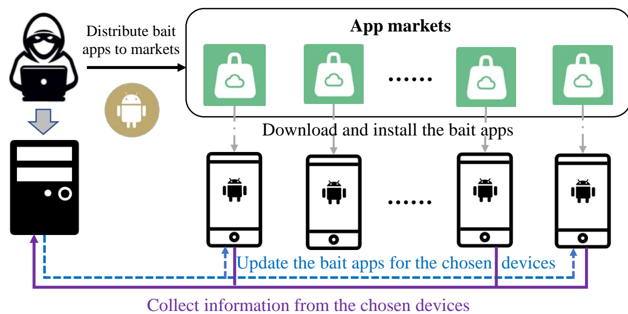


Fig. 2 VenomAttack scheme

date malicious payloads later. This could enable the resulting apps to be spreaded to a large user base. DICI, proposed by Gao *et al.* [19], allows apps to access and invoke functionalities implemented in other apps using official Android APIs. Thus, it is possible to quickly develop a customized hotpatch framework by leveraging DICI, where the bait app downloads a malicious payload as a new app and executes the payload via the DICI mechanism. We leave them as future work.

4 Our Attack: VenomAttack

In this section, we first present the scheme and workflow of VenomAttack and then provide the details of VenomAttack.

4.1 Threat Model and Attack Scheme

The attack scheme of VenomAttack is depicted in Figure 2. The adversary develops some bait apps with different interesting and/or useful functionalities to attract more users and distributes them via app markets (e.g., Google Play Store). In contrast to prior attacks such as [3–6, 8], which assume that the bait app is a malware and contains an attack payload, our bait app (before loading the patch file) does not contain any attack payloads except for its normal functionalities and a hotpatch technique. We also assume that the bait app requires the INTERNET permission and declares the required task affinities of target apps in the manifest file, both of which are widely-required in existing attacks [3, 5, 6, 20].

An app with a hotpatch technique is arguably harmless. To justify this, we implemented a bait app with a hotpatch framework, called EasyNote. The bait app was checked by a holistic malware hunting platform VirusTotal which consists of 63 app scan engines. No engines classified the bait app as a malicious one. We also examined several app markets including Google Play Store, Xiaomi App Store, Ali App Distribution Platform, Huawei App Gallery, and Apkpure. We found only the Google Play Store declared in its Developer Program Policy that all apps are not allowed to be modified, replaced, or updated by themselves in any way other than Google Play's update mechanism while others did not declare. However, the bait app was successfully released on the Google Play Store on September 25, 2020 after approximately 5 days of submission, and successfully updated by removing some useless permissions on March 17, 2021 after approximately 20 minutes of submission. This means that Google either does not check whether apps have hotpatch techniques or fails to detect the presence of the hotpatch technique in our app. Furthermore, we investigated 250 popular apps from 5 app markets, out of which 108 apps use hotpatch frameworks. This indicates that hotpatch frameworks are widely used by benign apps and cannot be used as a key indicator for identifying our attack. We emphasis that without using hotpatch techniques, apps with activity hijacking payloads would be detected by existing detection tools before being installed by users [6, 8, 10, 12, 13].

All the devices installed the bait app and any target apps could be the victims of VenomAttack. Once the bait app gets started on a device, it can collect various information such as IP address via the class `InetAddress`, device brand via the class `Build` in `android.os`, resolution of the screen via the class `DisplayMetrics`, and the list of installed apps with their versions. By leveraging these information, the adversary can identify vulnerable devices and dynamically choose one or more devices and apps to attack. For instance, the adversary can choose specific devices to attack, e.g., devices from some specific countries by designated IP address ranges, devices produced by some specific companies. Furthermore, by leveraging hotpatch techniques, VenomAttack can carry out a spectrum of customized attacks on a large scale by updating attack payload into the bait apps, without re-installation and re-distribution. Although the bait app contains the attack payload after hotpatching, it cannot be detected by the existing tools [6, 8, 10, 12, 13], as it has already been installed on the devices. The attack payload could be removed over a prolonged time period, e.g., a successful attack, making the attack less suspicious.

We explore how to launch activity hijacking attacks in Android using the proposed attack scheme, as activity hijacking is one of the most powerful attacks and can be used to implement various attacks. Furthermore, most popular apps by default are vulnerable to activity hijacking attacks.

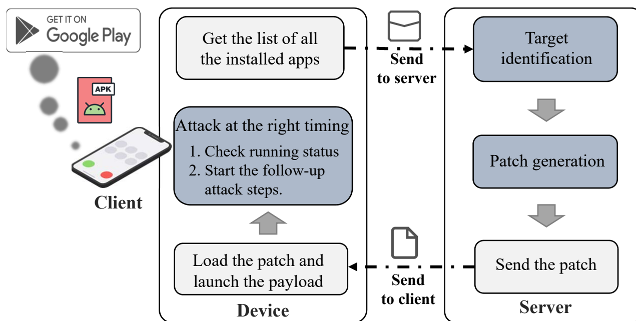


Fig. 3 Workflow of VenomAttack

4.2 Workflow of VenomAttack

In this subsection, we explore how to launch VenomAttack. Figure 3 depicts the workflow of conducting such an attack, including three main phases: **(1)** ***Target identification***, which identifies and selects vulnerable devices to attack after obtaining the installed apps via the bait app; **(2)** ***Patch generation***, which creates a patch file including the attack payload for each target app and device; and **(3)** ***Attack at the right timing***, which checks if the target app is running and then hijacks the user interaction at the right timing.

VenomAttack works as follows. When the bait app is launched by the user, it first collects the list of apps installed on the device and sends the list to the adversary-chosen server. The adversary then can choose any vulnerable target (i.e., app and device) from the list where the task affinity has been declared in the bait app. For each selected target app, the server creates a patch file from the original bait app and a new bait app updated with an attack payload. Later, the patch file is sent back to the victim. When the bait app is launched again by the user, it checks the existence of the patch file. If it has received a patch file, the bait app automatically loads the patch file to enable the attack mechanism. Then, the bait app has the ability to attack the target app and starts the follow-up attack steps. The entire process of VenomAttack is automated and stealthy to the user, while the adversary only needs to select vulnerable apps and devices to attack.

4.3 Illustrating Example

We illustrate the workflow of VenomAttack with a phishing attack in Figure 4, where the subfigures (a–d) show the GUIs of the running apps, and the subfigures (1–4) show the status of the task stack. The goal is to conduct a phishing attack against an *Anonymous* financial app. We now assume that the patch file has been loaded by the bait app.

When the bait app starts, it checks if the target app is running in the background or not. If the target app is running, as shown in Figure 4(a) and Figure 4(1), it launches a designed transparent activity with the same affinity as the target and the `singleTask` launch mode, by which the transparent activity will be pushed onto the target app's task in the foreground.

To be less suspicious, the foreground activity should not change the user interaction from the perspective of the user. Thus, the lifecycle of the transparent activity is carefully manipulated as shown in Figure 5 such that the transparent activity immediately moves back into the background at its first creation. We found that the API functions `onCreate`, `onStart` and `OnResume` are successively invoked at the first creation of an activity. Thus, we can invoke the API function `moveTaskToBack` in any of them, by which the transparent activity is moved back into the background (i.e., on the *pause* state first, then entering the *stop* state). This ensure that the user interaction is not changed from the perspective of the user, but in fact, the top of the target app's task in the background becomes the transparent activity as shown in Figure 4(b) and Figure 4(2). Remark that `onStart` and `OnResume` are also invoked when an activity resumes. Therefore, `moveTaskToBack` is only invoked in `onStart` or `OnResume` at the first creation.

When the user backs to the target app, the transparent activity resumes and comes to the foreground, i.e., moving from the *stop* state to the *running* state, as shown in Figure 4(c) and Figure 4(3). The real GUI layout of the foreground task is shown in Figure 6. Our goal is to hijack the user interaction of the target app by a malicious activity when the user backs to the target app, which is the right attack timing. We found that the API functions `onRestart`, `onStart` and `OnResume` are successively invoked when an activity resumes. To achieve our goal, when the transparent activity resumes, we can immediately destroy the transparent activity and launch the fake login activity in `onRestart` or `onStart` or `OnResume`. From the user's point of view, this fake login interface is invoked by his/her own behavior, as illustrated in Figure 4(d) and Figure 4(4). We argue that the user likely believes that the target app is designed in this way as it is a common phenomenon that many login-required apps would require users to re-login again if the session is expired (cf. [21]). Thus, the user will "normally" enter his/her account and password in the fake login interface, which will be stolen and sent back to the server by the bait app. From the user's perspective, he/she just performed a normal login operation. Remark that `onStart` and `OnResume` are also invoked at the first creation of an activity, thus, we can only

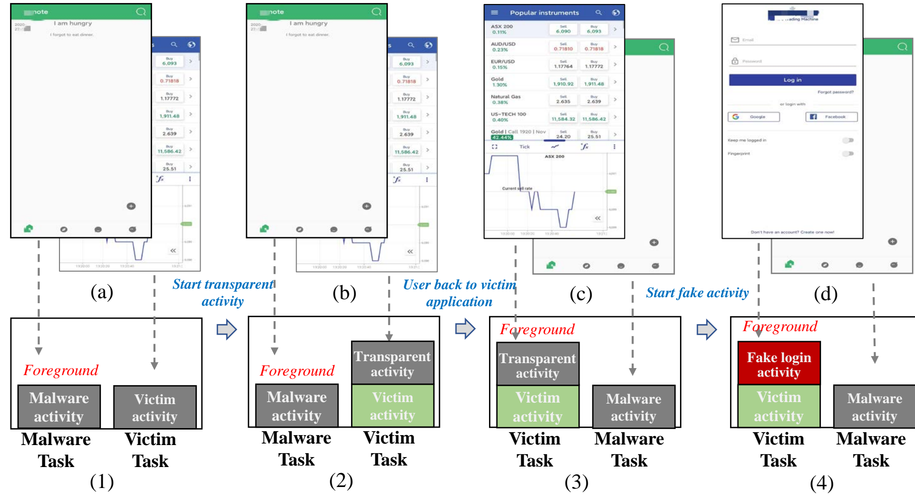


Fig. 4 Illustrating example of VenomAttack. (a), (b), (c), (d) show the GUIs from the user's perspective, where the top GUI is in the foreground and the rest are in the background. (1), (2), (3), (4) show the status of the corresponding task stacks.

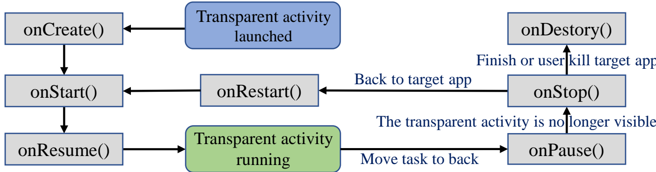


Fig. 5 Lifecycle of the transparent activity

destroy the transparent activity and launch the fake login activity when the transparent activity resumes.

4.4 Target Identification

Existing activity hijacking attacks usually plan out their attack strategies against specific devices that have installed the malicious app and target app, and hence do not have the chance to dynamically choose target apps and devices. To make the attack more powerful and practical, our attack can dynamically choose one or more vulnerable devices, according to the devices and installed apps therein. To achieve this, the bait app is implemented to obtain the installed apps in each device via either the API function `getInstalledPackages` or the API function `getInstalledApplications`, and the resolution of the victim. The resolution is critical to generate a malicious activity so that its GUI matches the resolution of the victim. The list of installed apps and the resolution are sent back to the adversary-chosen server. All these steps only require the INTERNET permission on Android 10 or earlier, which is widely used in normal apps and is granted automatically at install time. To further reduce network traffic, the bait app can filter out of non-interesting apps, e.g., system apps, before sending the list back to the server. After that, the adversary can choose attack target (i.e., app and device).

Fig. 6 Layout of GUIs. (a) is the interface seen by the user and (b) is layout of GUIs.

4.5 Patch Generation

For each target app, the adversary has to create a patch file to mount an attack. However, it is non-trivial to automatically and efficiently generate a fake activity for patching as its source code is often unavailable and their binary code is usually obfuscated to protect IPR, while patches can be created from scratch for other attacks. Therefore, we present an automated fake activity generation approach to carry out

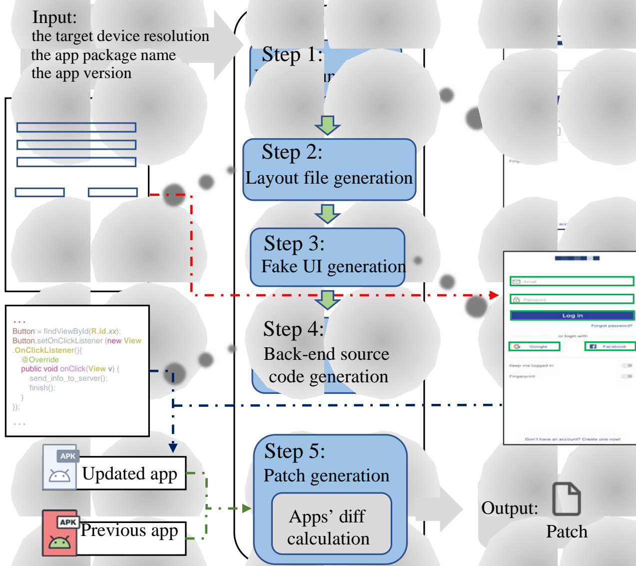


Fig. 7 Workflow of automated fake activity generation

large-scale phishing attacks in practice. Given the name and version of the target app and the resolution of the device (obtained via the bait app), our approach works as follows (cf. Figure 7).

Step 1, we launch the same app on a simulator with the same resolution. Then, we take a screenshot of the login UI of the target app, which is used as the background image of the fake UI, i.e., the upper-right image in Figure 7.

Step 2, we use UIAutomator to extract the number, length, width, and coordinate information of the interactive widgets (e.g., EditText and Button) in the login UI of the target app. After obtaining the detailed layout information, we create a layout file with the extracted widgets and add necessary file headers and file tails to ensure its usability. The UI looks like the upper-left image in Figure 7.

Step 3, we create the fake UI by putting the widgets onto the background image, resulting in the lower-right image in Figure 7, where the interactive widgets are highlighted with blue boxes. Note that the blue boxes are only used for illustration, they do not appear in real-attack scenarios.

Step 4, we create the back-end source code to handle the interactive widgets in the fake UI so as to obtain the user information after they input their credentials. For each EditText, the input text is saved when an input is detected. For each Button (e.g.,"Login"), we add a listener to check whether it is clicked and whether there is valid text in the associated EditText. After the user clicks the Button, the saved input text will be automatically sent back to the adversary-chosen server. All the widgets and back-end source code are connected via widget IDs defined in the generated layout file. We only create the source code for the relevant interactive widgets (e.g., Button, EditText) since they are the key and actionable widgets to obtain data from user input.

Step 5, we create an app from the base app, the fake activity and other attack code, resulting in an updated app. Then, we use the server-side hotpatch framework to automatically generate a patch file from the updated and base apps. The patch file will be sent to the bait app installed on the victim and automatically loaded once the bait app is launched. The underlying techniques used for hotpatching vary in hotpatch frameworks, which is beyond the scope of this work.

4.6 Attack at the Right Timing

Since it is suspicious to abruptly inject a malicious activity into the foreground without user interaction. However, it is non-trivial to determine a right attack timing, as third-party apps have been restricted from obtaining real-time system information via officially reserved APIs.To solve this problem, we present a new flaw in Android and a new bug in Android derivative systems (i.e., EMUI, MIUI, and Magic UI), identified through our systematical investigation. Each of them can be used to infer if a target app is running in the background or not, by which we can determine the right attack timing via the transparent activity as described in Section 3.

The flaw comes from the public API function `isTaskRoot` which returns whether the activity is the root of its task. `isTaskRoot` is often used to prevent multiple instances of an activity when it is launched with different intents. However, we found that it can be used to infer if the target app is running or not by invoking `isTaskRoot` after the creation of the malicious activity. If `isTaskRoot` returns `true`, there is only the malicious activity in the task, by which we can deduce that the target app is not running. Otherwise, it is running. We have reported the flaw to the Android Security Team of Google which is currently under processing.

The bug comes from the `ApplicationInfo` class. An `ApplicationInfo` instance has a public `int` type member variable, called `flags`, which provides some useful information such as category and stopped state. For example, `flags&FLAG_SYSTEM` indicates if the app is a system or third-party app, and `flags&FLAG_STOPPED` indicates if the app is in the force stopped state. `flags&FLAG_STOPPED` is cleared once the user starts the app after the app's initial installation (i.e., the default value is `true` after installation) and can be reset through the

"Force Stop" option in the Settings app. It is originally used to indicate if the app should receive broadcasts. However, after an in-depth investigation, we found that the value of `flags&FLAG_STOPPED` in these Android derivative systems is also affected by other actions (e.g., kill an app via the back stack, or start an app by clicking the icon). This means that `flags&FLAG_STOPPED` actually indicates whether an app is running or not. However, in official Android, the value of `flags&FLAG_STOPPED` is not affected by such actions. We have reported the bug to Huawei Bug Bounty Program and Xiaomi Security Center. The bug reported to Xiaomi Security Center has been confirmed, while the another one is still being processed. We suspect that this bug is introduced due to misunderstanding the semantics of `FLAG_STOPPED` literally when customizing the Android systems.

5 Evaluation

In this section, we first analyze the popularity of hotpatch frameworks in apps in the wild, then we evaluate VenomAttack from the following five aspects: compatibility of the hotpatch framework, effectiveness of running status checking via the newly-discovered flaw and bug, efficacy of our automated fake activity generation, ability to bypass the state-of-the-art defense mechanisms that were proposed to defend against activity hijacking attacks, and proof-of-concept attack examples. We used a PC machine with 64-bit Windows 10, Intel Core i5-8400, 2.8GHz and 32GB RAM as the server in our experiments.

5.1 Popularity of Hotpatch Frameworks

We selected top 10 apps in 5 different app categories listed in Table 1 from 5 Android app markets including the official market (i.e., Google Play Store) and four main third-parties (i.e., Apkpure, Xiaomi App Store, Ali App Distribution Platform, and Huawei App Gallery) as our analysis subjects. Therefore, there are 250 apps. Note that, the top 10 apps of each app category may be different due to the different ranking criteria between markets, but most of them are repetitive. We do not remove repetitive apps, as it is interesting to understand the difference of the same apps in different markets.

Due to the complexity and diversity of hotpatch techniques, it is non-trivial to check if these 250 apps use some hotpatch frameworks. Therefore, we use Androguard to analyze these 250 apps and check if each app uses some hotpatch framework by leveraging the keyword features of different hotpatch frameworks such as the name of the frameworks and specific API functions. When it finds that an app potentially uses a hotpatch framework, we manually analyze the app for further confirmation. Therefore, the number of apps that actually use hotpatch techniques may be far larger than our statistical study. In this study, the hotpatch frameworks we considered are shown in Table 1.

The results are reported in Table 1 for each market and each app category. Note that the order of apps in each app category may not reflect the exact order of apps in the corresponding market, as we manage to put the same apps from different markets in the same rows for ease of reference.

From Table 1, we find that 108 apps out of 250 apps are using hotpatch frameworks, resulting in a 43.2% usage rate. This indicates that the number of apps that are using hotpatch frameworks is very large in the wild, which makes it impossible to determine if an app is a malicious app by identifying hotpatch frameworks. In detail, 60% of apps from each market of Xiaomi App Store, Ali App Distribution Platform, and Huawei App Gallery are using hotpatch frameworks, and 26% of apps from the market Apkpure are using hotpatch frameworks. From Google Play Store, there are 5 apps out of 50 apps are using hotpatch frameworks, resulting in 10%. This indicates that some app developers obey the Developer Program Policy of Google. However, the successful release of EasyNote reveals that Google Play Store does not have a systematical approach to limit hotpatch frameworks used in real Android apps.

Besides the above results, we can also observe some insightful and interesting phenomenons from Table 1. Firstly, the same app may use different hotpatch frameworks or have different usage strategies for different app markets. For example, the app named *Momo* from Apkpure (in the social category) does not use any hotpatch frameworks, while the hotpatch framework Tinker is used in the apps with the same name from the three markets Xiaomi App Store, Ali App Distribution Platform and Huawei App Gallery. Actually, these apps are released by the same company. Similarly, the app named *Tantan* from Google Play Store (in the social category) does not use any hotpatch frameworks, while the apps with the same name from the other four third-party app markets use Tinker. Similar phenomenons can also be observed on the apps named by

- *Toutiao*, *Weico*, and *Xiaohongshu* (in the social category);
- *iQIYI*, *Tiktok*, *Bilibili*, and *Youku* (in the audio and video

Table 1 Hotpatch frameworks in apps in 5 Android app markets

App Category	Apkpure	Hotpatch Framework	Xiaomi App Store	Hotpatch Framework	Ali App Distribution Platform	Hotpatch Framework	Huawei App Gallery	Hotpatch Framework	Google Play	Hotpatch Framework
Social	**Wechat**	Tinker	**Wechat**	Tinker	**Wechat**	Tinker	**Wechat**	Tinker	**Josh**	-
	Momo	-	**Momo**	Tinker	**Momo**	Tinker	**Momo**	Tinker	**Investing**	-
	Tantan	Tinker	**Tantan**	Tinker	**Tantan**	Tinker	**Tantan**	Tinker	**Tantan**	-
	Weico	Robust	**Weico**	Robust	**Weico**	Robust	**Weico**	Robust	**Weico**	-
	Amazon Alexa	-	**QQ**	-	**QQ**	-	**QQ**	-	**QQ**	-
	Facebook	-	**Zhihu**	Tinker	**Zhihu**	Tinker	**Zhihu**	Tinker	**Tinder**	-
	LINELite	-	**Soul**	-	**momobeidanci**	-	**Soul**	-	**Soul**	-
	Xiaohongshu	Tinker	**KFC**	-	**Voov Meeting**	Tinker	**Xiaohongshu**	Tinker	**Xiaohongshu**	-
	YouTube	-	**Game Helper**	Tinker	**Zhenai**	-	**Zhenai**	-	**YouTube**	-
	Netflix	-	**Baidu Tieba**	-	**Toutiao**	Robust	**Toutiao**	-	**Toutiao**	-
Audio and Video	**iQIYI**	Dexposed	**iQIYI**	Tinker	**iQIYI**	Tinker	**iQIYI**	Tinker	**iQIYI**	-
	TinkerLite	-	**Youku**	Nuwa	**Youku**	Nuwa	**Youku**	Nuwa	**Youku**	-
	Tiktok	Dexposed	**Tiktok**	Robust	**Tiktok**	Robust	**Tiktok**	Robust	**Gmail**	-
	WeTV	-	**QQLive**	-	**QQLive**	-	**QQLive**	-	**DiDi**	-
	Instagram	-	**QQMusic**	Tinker	**QQMusic**	Tinker	**QQMusic**	Tinker	**Instagram**	-
	Yahoo Weather	-	**Bilibili**	Tinker	**Bilibili**	Tinker	**Bilibili**	Tinker	**Bilibili**	-
	WhatsApp Messenger	-	**Kwai Extreme**	Tinker	**Watermelonvideo**	Dexposed	**Watermelonvideo**	Dexposed	**WhatsApp Messenger**	-
	Weread	Hotfix	**NeteaseCloudMusic**	Tinker	**NeteaseCloudMusic**	Tinke	**Douyu**	-	**Twitch**	-
	Ctrip	Hotfix	**Kugou Music**	Tinker	**Noad**	-	**Ctrip**	Hotfix	**Moj**	-
	Twitter	-	**Kwai**	Tinker	**Jianying**	Robust	**Kwai**	Tinker	**Twitter**	-
Tools	**Dianping**	Robust	**Dianping**	Robust	**Dianping**	Robust	**Dianping**	Robust	**Dianping**	Robust
	Amap	-	**Dingdong**	-	**Amap**	-	**Amap**	-	**Amap**	-
	MiHome	Tinker	**Mi Home**	Tinker	**QQ Browser**	Tinker	**QQ Browser**	Tinker	**U-Mobile**	-
	Urban Company	-	**Anjuke**	Tinker	**UC Browser**	Tinker	**UC Browser**	Tinker	**UC Browser**	-
	Baidu Map	-	**Lianjia**	Robust	**Wesing**	Tinker	**Oasis**	-	**Uber**	-
	McDonald	-	**Meituan**	Robust	**Thuner**	-	**Thuner**	-	**Meituan**	Robust
	FamilyMart	-	**Meituan Takeout**	Robust	**Meituan Takeout**	Robust	**HMS Core**	-	**Zoom**	-
	Pinterest	-	**Eleme**	Hotfix	**Eleme**	Hotfix	**Dragonfly FM**	Tinker	**Pinterest**	-
	QQMail	-	**58.com**	Hotfix	**Traffic Control12123**	-	**58.com**	Hotfix	**QQMail**	-
	Google Map	-	**Douban**	-	**Homework Group**	-	**Yidui**	Tinker	**Douban**	-
Financial	**Alipay**	Andfix	**Alipay**	Andfix	**Alipay**	Andfix	**Alipay**	Andfix	**Alipay**	-
	Open Point	-	**Jingdong Finance**	Tinker	**Jingdong Finance**	Tinker	**Jingdong Finance**	Tinker	**PhonePe**	-
	Investing	-	**BOC**	-	**BOC**	-	**BOC**	-	**Paytm**	-
	HuobiGlobal	Tinker	**CMBC**	-	**Ding Talk**	-	**CMBC**	-	**HuobiGlobal**	Tinker
	Yahoo Finance	-	**UnionPay**	Tinker	**UnionPay**	Tinker	**WPS**	-	**UnionPay**	-
	Qunar	Hotfix	**PSBC**	-	**PSBC**	-	**Qunar**	Hotfix	**Qunar**	-
	Amazon Shopping	-	**ICBC**	-	**Baidu**	-	**Baidu**	-	**eBay**	-
	Google Pay	-	**ABC**	-	**Himalaya**	Tinker	**Himalaya**	Tinker	**YONO SBI**	-
	MoneyBack	-	**CCB**	-	**Huya**	Tinker	**Tiantian Fund**	-	**Mercado Libre**	-
	PayPal	-	**Wopay**	-	**Baidu Map**	-	**Douyin**	Robust	**Baidu Map**	-
Shopping	**Taobao**	-	**Taobao**	-	**Taobao**	-	**Taobao**	-	**Taobao**	-
	Jingdong	-	**Jingdong**	Tinker	**Jingdong**	Tinker	**Jingdong**	Tinker	**Jingdong**	Hotfix
	Tmall	-	**Tmall**	-	**Tmall**	-	**Tmall**	-	**Tmall**	-
	Pixiv	-	**Pinduoduo**	Tinker	**Pinduoduo**	Tinker	**Pinduoduo**	Tinker	**Pinduoduo**	Tinker
	MangoMall	-	**Alibaba**	-	**Alibaba**	-	**Pingan Securities**	-	**Alibaba**	-
	Fliggy	-	**Zhuanzhuan**	Tinker	**Baihe**	-	**Fliggy**	-	**Fliggy**	-
	HKTVmall	-	**Dewu**	Robust	**Suning**	Robust	**Suning**	Robust	**Wish**	-
	HongKongMovies	-	**CR Vanguard**	-	**Dangdang**	Robust	**Dangdang**	Robust	**SHEIN**	-
	YuuReward	-	**VIPshop Holdings**	-	**Amazon Shopping**	-	**VIPshop Holdings**	-	**Flipkart**	-
	ViuTV	-	**Idlefish**	-	**Idlefish**	-	**BoCom**	Dexposed	**Lazada**	-

Note 1: We consider 5 categories in each app market, where the top 10 apps of each category may be different due to the different ranking criteria between markets.
Note 2: The hotpatch frameworks include Tinker, Robust, Andfix, Nuwa, Hotfix, and Dexposed.

category);

- *UC Browser* (in the tools category);
- *Alipay*, *UnionPay*, and *Qunar* (in the financial category);
- *Jingdong* (in the shopping category).

The difference of hotpatch frameworks for the same app may be due to the fact that app markets adopt different policies. As mentioned in Section 4, there is no restriction on self-updating of an app released on Apkpure, Xiaomi App Store, Ali App Distribution Platform, and Huawei App Gallery. In contrary, Google Play Store disallows third party apps to be modified, replaced or updated by themselves. Actually, the developers of hotpatch frameworks are also aware of app markets' policies. For instance, the instruction document of Tencent Bugly, an exception reporting and statistics platform released by Tencent, states that due to Google Play's policy restrictions, an app using the Tinker framework may be detected as a violation and hence removed or even be banned. Therefore, the same app may adopt different usage strategies and different hotpatch frameworks to circumvent the review process of different app markets in real-world scenarios.

Secondly, Tinker is the most frequently used hotpatch framework in these 250 apps, while Robust the second most frequently used one. This is consistent with our previous survey results which are mentioned in Section 2, namely, Tin-

Table 2 Results of compatibility and running status collection

Android Device	Android Version	API Level	Result of Compatibility	Result of Flaw	Result of Bug
Google Pixel 2	Android 7	24	Succeeded	Succeeded	Failed
Google Pixel 2	Android 7.1	25	Succeeded	Succeeded	Failed
Google Pixel 2	Android 8	26	Succeeded	Succeeded	Failed
Google Pixel 2	Android 8.1	27	Succeeded	Succeeded	Failed
Google Pixel 2	Android 9	28	Succeeded	Succeeded	Failed
Google Pixel 2	Android 10	29	Succeeded	Succeeded	Failed
HUAWEI Nova5 Pro	EMUI 9.1.1	28	Succeeded	Succeeded	Succeeded
HUAWEI Nova5 Pro	EMUI 10.1.0	29	Succeeded	Succeeded	Succeeded
HUAWEI HONOR 30S	Magic UI 3.1.1	29	Succeeded	Succeeded	Succeeded
Xiaomi Redmi10X Pro	MIUI 11.0.5	29	Succeeded	Succeeded	Succeeded
Xiaomi Redmi K30	MIUI 11.0.15	29	Succeeded	Succeeded	Succeeded

ker and Robust have the best compatibility and are both open sourced. Furthermore, we also find that although some hotpatch frameworks (such as Andfix, Hotfix, and Dexposed) are no longer updated, they are still used in some apps, meaning that the developers of these apps are still maintaining these frameworks to be compatible with the recent versions of Android.

5.2 Compatibility Evaluation

As bait apps play the most important role in VenomAttack, we evaluate the compatibility of VenomAttack by deploying the bait app EasyNote on the recent Android versions.

Setup. We use the recent 6 official and 5 recent commercial versions of Android as shown in Table 2. As reported by StatCounter, the cumulative adoption of the 6 official versions is 82.91% on Feb. 2021. For each official version of Android, we create an emulator using Google Pixel 2. The 5 commercial versions are Android mobile devices. We install EasyNote on the emulators and devices, and send patch files containing new classes, methods, and resource files to EasyNote to investigate if the hotpatch can succeed.

Results. The results are reported in Table 2, where Succeeded means that hotpatch was succeeded in our experiments, otherwise Failed. We can see from the fourth column that EasyNote can successfully load all the patch files including the one that contains Java classes, methods, *.so* library files, and resource files at the same time, on all the Android devices. This demonstrates that bait apps could be updated by leveraging hotpatch techniques in recent official versions and commercial versions of Android in real-world scenarios. Therefore, hotpatch techniques can be used in VenomAttack.

5.3 Evaluation of Running Status Identification

For activity hijacking attacks, it is important to determine a right attack timing [3]. We present a flaw and a bug for checking the running status of the target app by which we can determine a right attack timing. Thus, we evaluate the effectiveness of the flaw and the bug, respectively.

Setup. We consider all the versions of Android as in Section 2 and installed 2 third-party apps in each emulator while each real Android mobile device has installed more than 10 third-party apps. We examine whether our bait app EasyNote can successfully infer the running status of these apps on all the Android versions.

Results. The results are reported in the last two columns in Table 2. We can observe from the column *Result of Flaw* that EasyNote successfully identified the running status of all the apps on all the Android versions by leveraging the flaw (i.e., API function `isTaskRoot`), without requiring any permissions. By exploiting the bug (i.e., the flag in the `ApplicationInfo` class), we successfully identified the running status of all the apps on all the derivatives of Android, while failed on all the official Android versions (cf. the last column in Table 2). This means that EasyNote can successfully infer whether the target apps are running or not on all these Android derivative systems. Although the bug does not affect the official Android versions, the use of Android derivative systems in the world has reached 48.52% on April 2021, the impact of the bug is still significant. To attack devices with official Android, the adversary has to leverage the flaw, while the adversary can use the flaw and/or the bug to attack devices with these Android derivative systems.

The breakthrough in the sandbox model of Android will bring severe security risks. For example, in VenomAttack, it can be used to determine a right attack timing. The bait app EasyNote can also check whether some security defense

apps are running so that it can bypass them, thereby broadens the attack capability of VenomAttack. We highlight that, both the flaw and the bug can be used by other activity hijacking attacks.

5.4 Evaluation of Fake Activity Generation

We highlight that VenomAttack is fully automated including the fake activity generation for phishing attacks. The key to success of a phishing attack is the high visual similarity between the fake and original UIs [22, 23]. Therefore, we evaluate the effectiveness and efficiency of fake UI generation through visual similarity comparison and the generation time.

Setup. We collect 50 popular apps from Google Play Store as our subjects in financial and social categories, as apps in these categories are usually security-/privacy-critical and contain login interfaces [22, 24]. The 25 financial apps are randomly selected from the top 100 financial apps with more than 1 million downloads, while the 25 social apps are randomly selected from the top 100 social apps with more than 10 million downloads. We evaluate using Xiaomi Redmi K30.

For each app, we create one fake login UI. To evaluate the effectiveness of our fake activity generation, we compare the visual similarity for each pair of the generated fake UI and the original UI, using two widely-used image similarity comparison algorithms: Cosine similarity and Structural similarity. The former one represents each picture as a vector and characterizes the similarity of two pictures by calculating the cosine distance between two vectors. The latter one is a full-reference image quality evaluation index that measures image similarity from three aspects: brightness, contrast, and structure. The values of both similarity comparison algorithms range from 0 to 1, where the larger the value, the more similar the UI pages. To evaluate the efficiency of our fake activity generation, we measure the execution time for each selected app, demonstrating the ability of VenomAttack to quickly deploy large-scale phishing attacks.

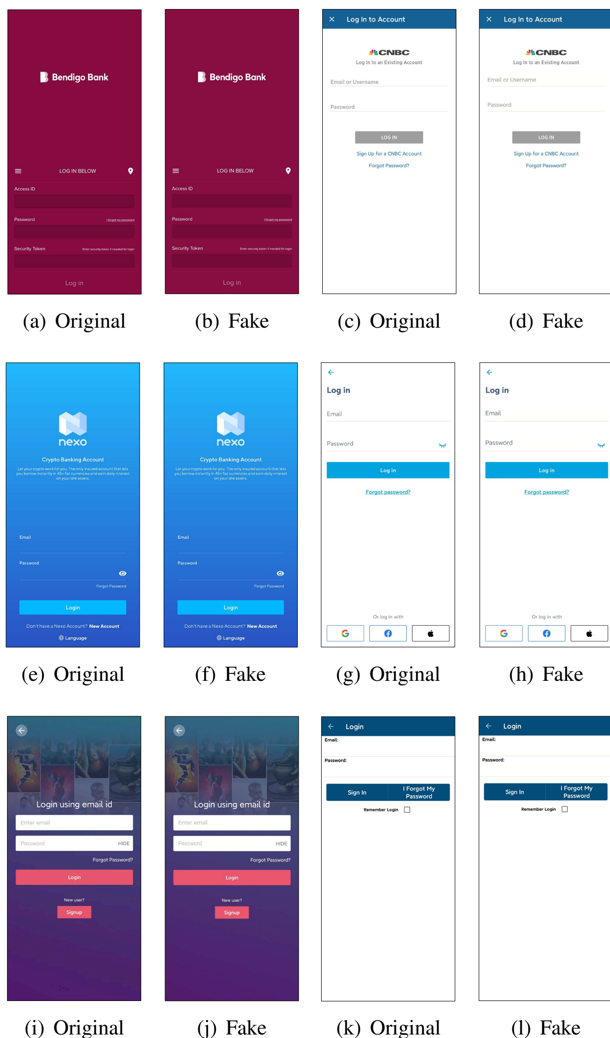
Table 3 Results of fake UI generation

No.	App Name	Cosine Similarity	Structural Similarity	Generation Time (sec)
1	**Alipay**	1.0000	0.9950	2.56
2	**AASTOCKS**	1.0000	0.9984	2.76
3	**AvaTradeGO**	0.9989	0.9925	3.07
4	**Bank of China**	1.0000	0.9922	2.55
5	**Bendigo Bank**	1.0000	0.9995	2.56
6	**BoC Pay**	1.0000	0.9935	2.78
7	**CMC Markets**	0.9941	0.9889	3.05
8	**CNBC**	1.0000	0.9886	2.58
9	**CommSec**	0.9999	0.9907	2.54
10	**Crypto**	1.0000	0.9949	2.56
11	**FCMB**	1.0000	0.9946	2.59
12	**HANGSENG BANK**	1.0000	0.9998	2.58
13	**HMRC**	1.0000	0.9974	2.72
14	**WireBarley**	1.0000	0.9934	2.58
15	**Inversting**	1.0000	0.9949	2.54
16	**Nexo Wallet**	1.0000	0.9983	2.62
17	**Plus500**	1.0000	0.9969	2.62
18	**Remitly**	1.0000	0.9926	2.67
19	**ShopBack**	1.0000	0.9947	2.57
20	**StockMarkets**	1.0000	0.9922	2.60
21	**Tiger Trade**	0.9998	0.9889	2.60
22	**TradingView**	1.0000	0.9898	2.60
23	**TransferWise**	1.0000	0.9929	2.68
24	**Wallet**	1.0000	0.9912	2.59
25	**REMIT**	1.0000	0.9934	2.65
26	**Facebook**	1.0000	0.9909	2.72
27	**Hago**	1.0000	0.9969	2.56
28	**Hello Yo**	1.0000	0.9958	2.59
29	**Houseparty**	1.0000	0.9905	2.56
30	**Jaumo**	1.0000	0.9990	2.54
31	**Josh**	1.0000	0.9996	2.66
32	**LivU**	1.0000	0.9971	2.55
33	**MeetMe**	1.0000	0.9933	2.66
34	**Mico**	1.0000	0.9982	2.56
35	**QQ**	1.0000	0.9933	2.59
36	**Roposo**	1.0000	0.9955	3.28
37	**SKOUT**	1.0000	0.9975	2.91
38	**Tagged**	1.0000	0.9943	2.59
39	**TANTAN**	1.0000	0.9947	2.57
40	**Telegram X**	0.9999	0.9962	2.54
41	**TikTok**	1.0000	0.9928	2.61
42	**Tumblr**	1.0000	0.9965	2.91
43	**Twitter**	1.0000	0.9959	2.57
44	**Viber**	1.0000	0.9959	2.63
45	**VidStatus**	1.0000	0.9960	2.58
46	**WeChat**	1.0000	0.9869	2.58
47	**Weico**	1.0000	0.9865	2.62
48	**WhatsApp**	1.0000	0.9979	2.56
49	**WhosHere**	1.0000	0.9994	2.55
50	**Xiaohongshu**	1.0000	0.9909	2.61

Results. The results are reported in Table 3. We can observe that the cosine similarity is: 1 for 45 apps out of 50 apps and very close to 1 for the other 5 apps. The structural similarity is: over 99% for 44 apps and very close to 99% for the other 6 apps. Figure 8 presents 6 randomly chosen pairs of the fake UIs and the original UIs, where the corresponding apps are: Bendigo Bank app, CNBC app, Nexo Wallet app, TransferWise app, Roposo app and WhosHere app. One can observe that it is very difficult to distinguish them visually.

The high similarity indicates that the fake UIs are similar enough to masquerade as the original ones, thus, difficult to be distinguished from the original UIs by users. In addition, the execution time of fake activity generation is close to 3 seconds, indicating that VenomAttack has ability to generate a large number of fake activities for real-world attacks in a very short time. This cannot be done manually.

There are few apps on which the similarity of the fake UI and the original UI is relatively smaller. To understand the



(a) Original (b) Fake (c) Original (d) Fake
(e) Original (f) Fake (g) Original (h) Fake
(i) Original (j) Fake (k) Original (l) Fake

Fig. 8 Comparison of original UIs and fake UIs

reason, we conduct case studies by manually checking these UIs. We found that the font setting of hints in some input widgets such as EditText cannot be correctly extracted, causing slight differences between the generated fake UIs and the original UIs. However, such slight differences do not affect VenomAttack in practice according to the results of our user study (cf. Section 6). Furthermore, this issue could be resolved by providing correct fonts only once for a target app.

5.5 Bypassing Defense Mechanisms

We choose the most representative and state-of-the-art defense mechanisms that were proposed to defend against activity hijacking attacks, to examine the other activity hijacking attacks mentioned in Section 3 as well as ours. The defense mechanisms have different strategies, including 1) offline analysis [6, 8, 10, 12, 13], 2) real-time detection [11, 14, 17], and 3) various Android design restrictions. The results are summarized in Table 4.

Offline analysis. We examine 5 state-of-the-art offline analysis methods against activity hijacking attacks [6,8,10,12,13]. Lee *et al.* [6] developed an analyzer which statically analyzes launch modes, task affinities and intent flow, to detect activity hijacking attacks (only *malware*⇒*target*) according to the rules in the operational semantics. Xiao *et al.* [8] developed an app, named TICK, to detect *malware*⇒*target* and *target*⇒*malware* attacks by checking a set of specified attack conditions. Luo *et al.* [10] designed a symbolic execution system, called Centaur, to check if there is a feasible path such that a selected activity resides in the same task as a victim activity. This work considered only *malware*⇒*target*. Liu *et al.* [13] presented a framework MR-Droid for ranking the risk of a given app by analyzing various inter-component communication (ICC) flow features, hence MR-Droid can detect activity hijacking attacks that utilize intent flags. TDroid [12] first statically slices an app into a set of runnable fragments where each fragment can launch one of its own activity in the background and may put the activity on top of a benign app's task, then repacks and dynamically executes each fragment to expose the malicious activity.

Attacks proposed in [4, 6–8] could be detected by the above methods, and the attacks proposed in [3, 9] that launches a fake activity in a new task to race the target app could be detected only by TDroid [12]. The *target*⇒*malware* attack of [5] cannot be detected by [6, 10, 12] and the attack of [5] without using intent flags cannot be detected by [13]. Stranghogg 2.0 that is executed through Java reflection could be partially detected by [12] only, as TDroid handles reflection partially while it is unclear if the others can handle reflection.

The bait apps used in VenomAttack does not contain any attack payloads before hotpatching, hence cannot be detected. The bait apps are hotpatched without re-installation and re-distribution, hence cannot be detected after hotpatching too, as these methods analyze apps before installation.

Real-time detection. We examine 3 state-of-the-art real-time detection methods: WindowGuard [11], ActivityShielder [14] and Activity Hijacking Protector [17], that were proposed to defeat activity hijacking attacks at runtime. Since these tools are not publicly available and non-trivial to re-produce, we only analyze them theoretically.

WindowGuard checks if: 1) the activities in each back stack that is part of the foreground activity session are all from the same app, where the foreground activity session is

Table 4 Results of bypassing state-of-the-art defense mechanisms

Attacks \ Defense	Offline Analysis Methods					Android Design Restrictions	Real-time Detection Methods		
	Lee *et al.* [6]	TICK [8]	CENTAUR [10]	MR-Droid [13]	TDroid [12]		WindowGuard [11]	ActivityShielder [14]	Activity Hijacking Protector [17]
Attack in [7]	●	●	●	●	●	●	●	●	◐
Activity hijacking [3]	○	○	○	○	●	●	●	●	●
Task hijacking [5]	◐	●	◐	◐	◐	Unknown	●	●	◐
ActivityHijacker [4]	●	●	●	●	●	●	●	●	◐
Information stealing attack [8]	●	●	●	●	●	●	●	●	◐
Activity injection [6]	●	●	●	●	●	●	●	●	○
Activity hijacking [9]	○	○	○	○	●	●	●	●	○
Stranghogg 2.0	Unknown	Unknown	Unknown	Unknown	◐	●	○	○	◐
VenomAttack	○	○	○	○	○	○	○	○	○

●: Fully detect ◐: Partially detect ○:Unable to detect

the ordered activities starting from the launcher activity, end with the activity in the foreground and each internal activity is started by the pervious one; 2) all the activities whose GUIs are visible belong to the same app or a whitelist of apps; and 3) activity transition in the foreground must be initiated by either the foreground app or an app from a whitelist, where 1) and 2) are checked when the foreground activity changes. If any of them is violated, WindowGuard warns users with a notification. Thus, WindowGuard can detect all the prior activity hijacking attacks except for Stranghogg 2.0. However, WindowGuard cannot detect VenomAttack. First, in our attack, either the back stack is not a part of the foreground activity session or all the activities in the back stack all from the same app, hence 1) is not violated. Second, when the transparent/malicious activity enters into the foreground, only the GUIs of the transparent/malicious activity and system activities are visible while system activities must be in the whitelist, hence 2) is not violated. Note that the visibility of a GUI of an activity is a state rather than whether it can be seen on the screen. Thus, although the GUI of the target activity covered by the transparent activity can be seen on the screen, it is indeed in the hidden state. Third, the activity transition that puts the transparent or malicious activity into the foreground is initiated by the bait app or a system activity while system activities must be in the whitelist, hence 3) is not violated.

ActivityShielder checks if each launching activity can be tracked back to the launcher of some trusted app. Except for Stranghogg 2.0, all the other attacks including VenomAttack will be tracked back to the app used for attacking. All the other attacks (except for VenomAttack) assume that the malware is installed on the victim while the malware could be detected by some of other methods (e.g., [25, 26]), thus this malware could be regarded as an untrusted app. In VenomAttack, both the transparent and malicious activities are tracked back to the bait app which is downloaded from some app market and installed by the user. The bait app does not contain any attack payloads before hotpatching, thus will be considered as a trusted app in ActivityShielder. After hotpatching, the trustiness of the bait app does not change in ActivityShielder.

Activity Hijacking Protector checks if two UI-similar activities from two different apps are launched in a very short time such that the malicious one is launched after the target one, but is displayed before the target one. This work addresses activity hijacking based phishing attacks only. Thus, activity hijacking attacks [4, 5, 7, 8] that do not necessarily having UI-similar activities, and the phishing attacks [5, 6, 9] that do not satisfy the time constraint, cannot be detected. VenomAttack satisfies neither the UI-similarity nor time conditions, hence cannot be detected.

System design restrictions. Various design restrictions have been deployed until Android 10 which can mitigate activity hijacking attacks well. In particular, Google restricted specific API functions for checking if a target app is running, and for launching activities in the background. The vulnerability used in Stranghogg 2.0 has been fixed as well. Except for [5] which does explicitly state how to check if a target app is running and how to launch activities in the background, all the other attacks will not work in Android 10 and 11. VenomAttack leverages the newly-discovered flaw to check if a target app is running (cf. Section 3) and uses a carefully designed transparent activity to inject malicious activities, hence circumventing all these restrictions. We will demonstrate it in user study (cf. Section 6).

5.6 Proof-of-Concept Attack Examples

VenomAttack is a very powerful attack scheme, enabling a spectrum of customized attacks on a large scale in real-world scenarios. It should be noted that for all the attack examples, the adversary can distribute bait apps via app markets

and dynamically choose vulnerable devices using the information collected by the bait apps. For each target app of a chosen victim which has installed a bait app, the server creates an updated version of the bait app with attack payload from which a patch file is computed and sent back to the bait app installed on the victim. After that, the bait app carries the attack payload for mounting desired attacks.

Now, we assume that the bait app has been updated with a desired attack payload. We demonstrate more attack examples utilizing VenomAttack and highlight the impact of these attacks.

Phishing attack. We discuss two alternative attack scenarios for phishing attacks via VenomAttack. The first scenario is detailed in Section 3. a) When the bait app finds that the target app is running, it launches a transparent activity which will be pushed onto the target app's task. Once the user backs to the target app, the transparent activity enters the foreground, but the user sees the same GUI of the target app as the user left. If the user interacts with the target app, he/she indeed interacts with the transparent activity, then a phishing activity is launched by the transparent activity which enters the foreground. b) The second scenario is similar to the above one except that the target app is not running. If the transparent activity finds that the target app is not running, it immediately destroys itself and launches the phishing activity with the same affinity as the target app, by which the phishing activity becomes the root activity of the target app's task. When the user launches the target app by clicking its icon, the phishing activity enters the foreground. In both cases, the user will be finally tricked to input credential on the fake login interface, and the credential entered by the user will be stolen.

Spoofing attack. Android attempts to achieve security by allowing apps to request permissions *only* from the foreground. The runtime permission model is deployed to provide users with increased situational context to help them with their permission decisions. VenomAttack can be used to implement spoofing attacks, asking for any runtime permissions such as SMS, photos, microphone and GPS, while pretending to be some target apps installed on the device. Specifically, the bait app declares all required permissions in the manifest file, including the ones to be obtained via spoofing attacks. We emphasize that only the INTERNET permission is obtained by the bait app at first. From the user's perspective, only basic permissions required for the functionalities of the bait app are granted. But actually, it can hijack the GUI interaction of the target app as done for phishing attacks. When backing to or launching the target app, the fraud activity enters the foreground which tricks the user to obtain the running permissions while pretending to be the target app. For example, the bait app uses a fraud activity to obtain the LOCATION permission by targeting a navigation app. The spoofing attack exploits users' functional requirements to defraud users' trust, so that almost all the running permissions can be obtained gradually to mount further attacks.

DoS attack. VenomAttack can also be used to implement DoS attacks, making target apps unusable. We discuss two alternative attack scenarios for chosen target apps. a) In the first scenario, when the transparent activity enters the foreground by resuming, it protects itself to be killed unless its task is closed. To prevent itself from being killed, the API function `onKeyDown` of the transparent activity can be rewritten, which is invoked when some key is pressed down by the user, but does nothing. Consequently, the user can see the GUI of the target app on the screen but cannot interact with it. b) In the second scenario, when the transparent activity enters the foreground by resuming, it immediately invokes the API function `moveTaskToBack` in `onRestart`, `onStart` or `OnResume`. This will move the transparent activity back into the background again so that the user will never see the GUI of the target app in the screen. In either scenario, the target app cannot be used anymore from the user's viewpoint, resulting in very bad user experience. Such attacks could be used to compete other apps that have similar functionalities.

6 User Study

To demonstrate VenomAttack in real-world scenarios, we conduct a user study to answer the following research questions using the first scenario of our phishing attack (cf. Section 6).

RQ1: Can VenomAttack effectively steal credentials?

RQ2: Is it reasonable to achieve activity hijacking via the transparent activity in VenomAttack?

RQ3: Are participants aware of VenomAttack?

These research questions evaluate whether VenomAttack is able to obtain users' credentials stealthily in real-world scenarios. For **RQ1**, we record the credentials of participants received from the bait app, then compute attack success rate. For **RQ2** and **RQ3**, we ask participants to finish a questionnaire. This study was approved by the Institutional Review Board of our institutes and conducted under a limited environment without posing any security threats to real users.

Table 5 Questionnaire of the user study

No.	Question	Options
Question-1	Do you think the financial apps provide more functionalities after login?	Yes or No
Question-2	Based on your past experience, is it common to re-login after app switching?	Yes or No
Question-3	Are you aware any attacks during the study?	Yes or No
Question-4	If yes, when do you think attacks occur?	
Question-5	If yes, what makes you aware of the attacks?	

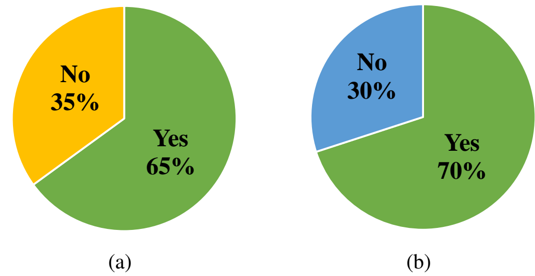


Fig. 9 Results of question-1 (a) and question-2 (b) in Table 5

6.1 Setup

Participants. We recruited 20 participants from our institutes whose ages range from 18 to 50. 10 of them are graduate students major in computer science and the others are colleagues in computer science too. To reduce noises of the study, all the participants have at least one year experience of Android mobile devices. We do not consider other factors such as gender and first language, that are negligible to our study.

Dataset. We randomly selected 10 popular apps as our subjects from the top 100 financial apps on Google Play Store. They are installed on 4 different Android devices (i.e., Xiaomi Redmi K30, Xiaomi Redmi 10X Pro, HUAWEI Honor 30S, and HUAWEI Nova5 Pro) which use derivatives of Android 10.0. The 10 financial apps are randomly partitioned into two groups: 5 target apps that will be attacked by VenomAttack, and 5 non-target apps that will be explored by the participants but will not be attacked.

Procedures. We first install our bait app EasyNote on the Android devices. For each participant, we randomly select one target app and one non-target app from the two groups, while each of 10 financial apps is constrained to be used 4 times. The bait app is hotpatched accordingly for each participant. We also provide each participant the corresponding accounts for the two financial apps and the bait app.

The user study begins with a brief introduction which asks each participant to do the following tasks without revealing the purpose of our study: **(a)** Opens the two financial apps and explores them with the provided accounts; **(b)** Opens the bait app and explores it with the provided account; **(c)** Re-explore the two financial apps with the provided accounts. During the above tasks, participants are asked to roughly record their actions and can stop at anytime when they smell a rat. During task **a**, no attack occurs. During task **b**, the bait app launches the transparent activity. During task **c**, the accounts are stolen and sent back to our server.

After finishing the above tasks, each participant is asked to complete a questionnaire. The questionnaire is shown in Table 5, where the second column shows the questions, and the third column shows the options for some questions.

6.2 Study Results

RQ1. We obtained the credentials of 19 participants, out of 20 participants, resulting in 95.0% attack success rate. We interviewed the participant to whom our attack fails, in order to understand the failure. We found that after the fake activity enters into the foreground during task **c**, instead of entering the account into the fake login interface, this participant clicks the Back button intended to explore the non-target app again. This clicking event pauses the fake activity, hence resulting in the failure of our attack. However, the participant was *not* aware of any abnormalities and did not report any attacks in **RQ3**. We argue that the bait app could mount the phishing attack again and again by launching the transparent activity when it is opened until the attack succeeds.

> **Answering RQ1.** VenomAttack achieves 95.0% attack success rate in our user study. The failed participant was *not* aware of any abnormalities and did *not* report any attacks in **RQ3**, thus VenomAttack could iteratively mount attacks until it succeeds in practice.

RQ2. Recall that in the first scenario of our phishing attack, the transparent activity resides on top of the target app's task and enters the foreground when the user backs to the target app. From the viewpoint of the user, he/she backs to the target app, after which the login interface is displayed. Therefore, it is important to study if it is reasonable to display the fake login interface via the transparent activity when the user backs to the target app, i.e., **RQ2**. To answer **RQ2**, we design two questions: question-1 and question-2 in Table 5. Question-1 studies the necessity of login and question-2 studies the commonality of re-login in practice.

The results are reported in Figure 9. 13 participants (65%) believe that the financial apps provide more functionalities after login, c.f. Figure 9(a). This means that users often have to login if they want to use more functionalities of financial

Table 6 Time cost of loading the transparent activity

Android Device	Android Version	API Level	Init Time (ms)	Back to Background Time (ms)
HUAWEI Nova5 Pro	EMUI 10.1.0	29	6	5
HUAWEI HONOR 30S	Magic UI 3.1.1	29	7	6
Xiaomi MI 9	MIUI 12.0.3	29	8	4
Xiaomi Redmi10X Pro	MIUI 11.0.5	29	7	3
Xiaomi Redmi K30	MIUI 12.0.5	29	17	7
OPPO Realme X	ColorOS V7	29	14	9

apps. 14 participants (70%) believe that it is common to re-login after app switching, c.f. Figure 9(b), meaning that there are many scenarios where users have to re-login to continue to use more functionalities of Android apps. This finding was also mentioned by Saltaformaggio *et al.* in Section 4.2.2 of [21]. These results indicate that achieving activity hijacking via a transparent activity will not draw users's attention.

We also analyze the time cost of landing the transparent activity on different Android devices, including the time cost for initializing the transparent activity and the time cost for moving the transparent activity to the background. The results are reported in Table 6. Although both time costs vary in Android devices due to the differences of performance between devices, the time cost for initializing the transparent activity is no more than 17 ms, and the time cost for moving the transparent activity to the background is no more than 9 ms. These results explain why participants are not aware of the transparent activity in our user study.

Answering RQ2: The results show that 65% of the participants (i.e., 13 participants) believe that the financial apps provide more functionalities after login and 70% of the participants (i.e., 14 participants) believe that re-login is common in practice. Thus, it is reasonable to achieve activity hijacking via the transparent activity.

RQ3. To answer **RQ3**, we ask participants to fill the question "*Are you aware any attacks during the study?*".

In our study, only *one* participant out of 20 participants is aware of the attack, however the account of this participant is also stolen by the bait app. To understand the reason, we interviewed this participant with two more questions: question-4 and question-5 in Table 5. The participant found that after submitting the account via the fake login interface, the fake interface is closed and the target app enters the foreground, but the login status of the target app does not change, namely, without login. Therefore, the participant considered if an attack occurred. Except for this participant, the other participants did not realize any security-related issues after completing the login. To understand the effect of this issue on stealthiness, we furthermore asked all the other 19 participants: "*Do you think the reason for the unchanged login status is a malfunction or security issue of the app?*". All the 19 participants thought it was the malfunction of the target app. We argue that even if a user is aware of the login status after submitting the account, the attack has already finished and the adversary actually has obtained his/her credential. It may be more stealthy to emit some messages to remind the user that the username or password is not correct and ask he/she to re-login again. This often occurs in practice.

Answering RQ3: Only *one* participant in our study considered if he/she was attacked due to the unchanged status after login. However, all the participants are *not* aware of any attacks before the credentials are stolen.

7 Discussion

7.1 Impact of VenomAttack

VenomAttack leverages hotpatch techniques, newly-discovered flaw and bug, and an elegant transparent activity. We also proposed a fully automated fake activity generation approach for adaptive large-scale phishing attacks. As examined in Sections 2 and 5, VenomAttack can circumvent the design restrictions of Android that are intended to mitigate activity hijacking attacks and can be successfully deployed on the recent versions of Android. Furthermore, hotpatch provides a convenient and effective way to implement dynamic code loading, which could be leveraged by other attacks. Both the flaw and bug can be used to infer if an app is running or not which could be used by other attacks. That would call more attention from Google.

VenomAttack can defeat the most representative and state-of-the-art defense mechanisms that were designated to defend against activity hijacking. This implies that the attack-and-defense game on activity hijacking is still far from over.

Compared over existing activity hijacking attacks, VenomAttack has many unique features, making it more powerful and stealthy. VenomAttack is shipped with a remote engine which can automatically generate attack payload including fake activities with high similarity of the original ones in a short time (c.f. Section 4). Together with hotpatch techniques, VenomAttack has the ability to mount sophisticated attacks on a large number of dynamically chosen target victims and target apps at the same time.

Our bait app EasyNote with the hotpatch technique Tinker was successfully released on Google Play Store and was not labeled as a malware by any of the 63 app security engines of VirusTotal. This reveals that Google and the security engines of VirusTotal either do not check whether apps have hotpatch techniques or fail to detect the presence of the hotpatch technique.

Our user study showed that VenomAttack can achieve very high attack success rate without users' awareness in the real scenarios, indicating that users should be more cautious about activity hijacking attacks when using Android apps.

7.2 Mitigation of VenomAttack

We discuss possible strategies and suggestions that could be adopted to mitigate VenomAttack.

For Android system, it is important and urgent to identify all the possible design weaknesses that could be used by the adversary to infer running information of apps and add more design restrictions on them. Indeed, though Android has added restrictions on the API functions that are used by prior activity hijacking attacks, our attack uses a newly-discovered flaw in Android or bug in derivatives of Android. On the other hand, hotpatch can be used to dynamically add attack payloads without re-installation and re-distribution. This issue could be mitigated by restricting installed apps to execute remotely downloaded code in Android with code integrity checks.

For the perspective of Android app markets, we recommend further enhancing the code audit for submitted apps. In our experiments, we successfully released an app with a hotpatch technique on Google Play Store. This implies that the official app market lacks of a systematic assessment method to detect the present of self-update in Android apps, although the Developer Program Policy states that it is not allowed to update in any ways other than Google Play's update mechanism. We believe that other app markets have the same issue, hence, their code audit should be enhanced as well.

App developers should pay more attention to the security of apps. According to our study, 1) the four basic components of apps should not be exported if it is not necessary [27]; 2) apps should protect their private-sensitive UI pages (e.g., the login interface of financial apps) to be screenshotted by the adversary so that the adversary cannot automatically construct fake UI pages [22, 28]; 3) private-sensitive UI pages should use secure keyboards which can helps the users aware if they are interacting with a fake UI page; and 4) apps should use toasts to remind users when they back to the background or enter the foreground, so that users can perceive if the current activity they are interacting is from the target app. Last but not least, users should also pay attention to toasts and keyboards.

7.3 On Activity Hijacking Attack in Android 11

We also verified our attack in Android 11 (API level 30). We found that Android 11 has attempted at countermeasures by implementing two mechanisms. First, a new permission QUERY_ALL_PACKAGES is introduced to control visibility of apps so that both the API functions `getInstalledPackages` and `getInstalledApplications` can only be invoked with this permission. However, it is a normal permission and is granted automatically at install time. Therefore, we could declared this permission in the manifest file. Second, two activities from two distinct apps cannot reside in the same task even the declared task affinities of the activities are same, as the real task affinity of an activity in back stack is the concatenation of the app's UID and the declared task affinity, where each app has a unique static UID. To circumvent this mechanism, the transparent activity could monitor the launching event of the target app via side-channel information [3, 9] although it is less reliable than directly using the newly-discovered flaw or bug. When the target app is launched, the transparent activity moves itself into the foreground by invoking the API function `moveTaskToFront`, then destroys itself and launches the malicious activity. Therefore, Android 11 is still vulnerable.

8 Conclusion

In this article, we proposed a novel activity hijacking attack VenomAttack, enabling automated, customized and large-scale attacks in recent Android versions. It is able to circumvent the sandbox security mechanism of Android and bypass the existing defense mechanisms including real-time detection. Our user study demonstrated the effectiveness of VenomAttack in the real-world scenarios without users' awareness, which would call more attention to the stakeholders. Finally, we discussed the impact of VenomAttack and proposed strategies and suggestions from the perspectives of Android system, app markets, and app developers to mitigate it.

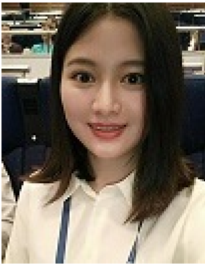
Acknowledgements This work was supported by the National Natural Science Foundation of China (Grant Nos. 62072309 and 6171101225).

References

1. Lu L, Li Z, Wu Z, Lee W, Jiang G. CHEX: statically vetting Android apps for component hijacking vulnerabilities. In: Proceedings of ACM Conference on Computer and Communications Security. 2012, 229–240
2. Gourdin B. Framing attacks on smart phones and dumb routers: Tap-jacking and geo-localization attacks. In: Proceedings of USENIX Workshop on Offensive Technologies. 2010
3. Chen Q A, Qian Z, Mao Z M. Peeking into your app without actually seeing it: UI state inference and novel Android attacks. In: Proceedings of USENIX Security Symposium. 2014, 1037–1052
4. Wang Z, Li C, Guan Y, Xue Y, Dong Y. Activityhijacker: Hijacking the Android activity component for sensitive data. In: Proceedings of International Conference on Computer Communication and Networks. 2016, 1–9
5. Ren C, Zhang Y, Xue H, Wei T, Liu P. Towards discovering and understanding task hijacking in Android. In: Proceedings of USENIX Security Symposium. 2015, 945–959
6. Lee S, Hwang S, Ryu S. All about activity injection: threats, semantics, and detection. In: Proceedings of IEEE/ACM International Conference on Automated Software Engineering. 2017, 252–262
7. Ren Y, Li Y, Yuan F, Zhang F. Hijacking activity technology analysis and research in Android system. In: Proceedings of International Conference on Trustworthy Computing and Services. 2013, 46–53
8. Xiao Y, Bai G, Mao J, Liang Z, Cheng W. Privilege leakage and information stealing through the Android task mechanism. In: Proceedings of IEEE Symposium on Privacy-Aware Computing. 2017, 152–163
9. Yang L, Zhi Y, Wei T, Yu S, Ma J. Inference attack in android activity based on program fingerprint. Journal of Network and Computer Applications, 2019, 127: 92–106
10. Luo L, Zeng Q, Cao C, Chen K, Liu J, Liu L, Gao N, Yang M, Xing X, Liu P. System service call-oriented symbolic execution of Android framework with applications to vulnerability discovery and exploit generation. In: Proceedings of Annual International Conference on Mobile Systems, Applications, and Services. 2017, 225–238
11. Ren C, Liu P, Zhu S. Windowguard: Systematic protection of GUI security in Android. In: Proceedings of Annual Network and Distributed System Security Symposium. 2017
12. Liu J, Wu D, Xue J. TDroid: exposing app switching attacks in Android with control flow specialization. In: Proceedings of IEEE/ACM International Conference on Automated Software Engineering. 2018, 236–247
13. Liu F, Cai H, Wang G, Yao D, Elish K O, Ryder B G. MR-Droid: A scalable and prioritized analysis of inter-app communication risks. In: Proceedings of IEEE Security and Privacy Workshops. 2017, 189–198
14. Yan F, Li Y, Zhang L. Activityshielder: An activity hijacking defense scheme for Android devices. In: Proceedings of International Conference on Computer Communication and Networks. 2018, 1–9
15. Chen S, Fan L, Chen C, Su T, Li W, Liu Y, Xu L. Storydroid: Automated generation of storyboard for android apps. In: Proceedings of IEEE/ACM International Conference on Software Engineering. 2019, 596–607
16. Chen T, He J, Song F, Wang G, Wu Z, Yan J. Android stack machine. In: Proceedings of International Conference on Computer Aided Verification. 2018, 487–504
17. Bkakria A, Mariem G, Cuppens-Boulahia N, Cuppens F, Lanet J L. Real-time detection and reaction to activity hijacking attacks in Android smartphones. In: Proceedings of International Conference on Privacy, Security and Trust. 2017, 253–258
18. Li L, Li D, Bissyandé T F, Klein J, Le Traon Y, Lo D, Cavallaro L. Understanding android app piggybacking: A systematic study of malicious code grafting. IEEE Transactions on Information Forensics and Security, 2017, 12(6): 1269–1284
19. Gao J, Li L, Kong P, Bissyandé T F, Klein J. Borrowing your enemy's arrows: the case of code reuse in android via direct inter-app code invocation. In: Proceedings of ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2020, 939–951
20. Tuncay G S, Qian J, Gunter C A. See no evil: Phishing for permissions with false transparency. In: Proceedings of USENIX Security Symposium. 2020, 415–432
21. Saltaformaggio B, Bhatia R, Gu Z, Zhang X, Xu D. GUITAR: piecing together android app guis from memory images. In: Proceedings of ACM SIGSAC Conference on Computer and Communications Security. 2015, 120–132
22. Chen S, Fan L, Chen C, Xue M, Liu Y, Xu L. GUI-Squatting attack: Automated generation of Android phishing apps. IEEE Transactions on Dependable and Secure Computing, 2019, 1–1
23. Song F, Lei Y, Chen S, Fan L, Liu Y. Advanced evasion attacks and mitigations on practical ML-based phishing website classifiers. International Journal of Intelligent Systems, 2021, 36: 5210–5240
24. Chen S, Su T, Fan L, Meng G, Xue M, Liu Y, Xu L. Are mobile banking apps secure? what can be improved? In: Proceedings of ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2018, 797–802
25. Song F, Touili T. Model-checking for android malware detection. In: Proceedings of Asian Symposium on Programming Languages and Systems. 2014, 216–235
26. Xu Z, Ren K, Song F. Android malware family classification and characterization using CFG and DFG. In: Proceedings of International Symposium on Theoretical Aspects of Software Engineering. 2019, 49–56
27. Chen S, Fan L, Meng G, Su T, Xue M, Xue Y, Liu Y, Xu L. An empirical assessment of security risks of global android banking apps. In: Proceedings of IEEE/ACM International Conference on Software Engineering. 2020, 1310–1322
28. Tang C, Chen S, Fan L, Xu L, Liu Y, Tang Z, Dou L. A large-scale empirical study on industrial fake apps. In: Proceedings of IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice. 2019, 183–192

Pu Sun is a Ph.D. student in ShanghaiTech University, China, supervised by Prof. Fu Song. He received the B.S. degree in Computer Science from Northeastern University at Qinhuangdao, China, in 2018. His research interests are in software testing and software security.

Sen Chen is an Associate Professor with the College of Intelligence and Computing, Tianjin University, China. Previously, he was a research assistant professor and postdoctoral research fellow at Cybersecurity Lab, School of Computer Science and Engineering, Nanyang Technological University from 2019 to 2020. He received his Ph.D. degree in computer science from East China Normal University, in 2019. His research focuses on software engineering, security, and data-driven analytics.



Lingling Fan is an Associate Professor with the College of Cyber Science, Nankai University, China. She received her Ph.D and B.S. degrees in computer science from East China Normal University, in 2019 and 2014, respectively. Previously, she was a research assistant professor and postdoctoral research fellow at Cybersecurity Lab, School of Computer Science and Engineering, Nanyang Technological University from 2019 to 2020. Her research focuses on program analysis and testing, Android application security analysis and testing.

Pengfei Gao is a Ph.D. student in ShanghaiTech University, supervised by Prof. Fu Song. He received the B.S. degree in Computer Science from China University of Mining and Technology in 2017. His research interests are in program analysis and software security.

Fu Song is an Associate Professor (Tenured) with ShanghaiTech University, China. He received the M.S. degree in Software Engineering from East China Normal University in 2009, and the Ph.D. degree in Computer Science from University Paris-Diderot in 2013. Previously, he was an Assistant Professor with hanghaiTech University from August 2016 to July 2021, lecturer and associate research professor with East China Normal University from August 2013 to July 2016. His research interests are primarily in formal methods and computer security.

Min Yang is a Professor and vice dean with School of Computer Science, Fudan University, China. He received the B.S. and Ph.D degrees in computer science from Fudan University in 2001 and 2006, respectively. His research interests are primarily in mobile security and privacy, AI security and privacy, and program analysis.