2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

Ghost Installer in the Shadow: Security Analysis of App Installation on Android

Yeonjoon Lee[1], Tongxin Li [2], Nan Zhang[1], Soteris Demetriou[3], Mingming Zha[4]
XiaoFeng Wang[1], Kai Chen[4], Xiaoyong Zhou[5], Xinhui Han[2], Michael Grace[5]
[1]*Indiana University, Bloomington* {yl52, nz3, xw7}@indiana.edu
[2]*Peking University* litongxin1991@gmail.com, hanxinhui@pku.edu.cn
[3]*University of Illinois at Urbana-Champaign* sdemetr2@illinois.edu
[4]*Institute of Information Engineering, Chinese Academy of Sciences* mmzha2013@gmail.com, chenkai@iie.ac.cn
[5]*Samsung Research America* zhou.xiaoyong@gmail.com, m1.grace@samsung.com

***Abstract*—Android allows developers to build apps with app installation functionality themselves with minimal restriction and support like any other functionalities. Given the critical importance of app installation, the security implications of the approach can be significant. This paper reports the first systematic study on this issue, focusing on the security guarantees of different steps of the *App Installation Transaction* (AIT). We demonstrate the serious consequences of leaving AIT development to individual developers: most installers (e.g., Amazon AppStore, DTIgnite, Baidu) are riddled with various security-critical loopholes, which can be exploited by attackers to *silently* install any apps, acquiring dangerous-level permissions or even unauthorized access to system resources. Surprisingly, vulnerabilities were found in all steps of AIT. The attacks we present, dubbed *Ghost Installer Attack* (GIA), are found to pose a realistic threat to Android ecosystem. Further, we developed both a user-app-level and a system-level defense that are innovative and practical.**

I. Introduction

Android dominated the mobile operating system (OS) market with an 87.6% share in the second quarter of 2016 [4]. The strength of Android is its open-source nature, which enables convenient customizations and adaption to different needs. However, with the blessing from its flexibility and decentralized management, comes the curse of fragmentation and confusion, which can have significant security impacts. No standards are available to control the security qualities of the system apps pre-installed by different device manufacturers and carriers, and no guidelines are available to inform the app developers precisely what the OS can protect and what should be taken care of by the developers themselves. Even for services as critical as *app installation*, all Android provides are nothing more than nuts and bolts (i.e., AOSP Download Manager, Package Manager) and on top of them, the developers and device manufacturers are supposed to build up their own services. The security implications of this treatment can be significant, which however has never been investigated before.

Security risks in app installation. On Android, an app can be installed or updated programmatically, with or without human interventions. More specifically, consider app installation as a *transaction*. At the center of it is an *installer* app (app with installation capability: e.g., appstore app) either with or without the `INSTALL_PACKAGES` permission. In the former case, the *installer* app installs an apps silently (without user interaction) by directly invoking the Package Manager Service (PMS), whereas in the latter case the *installer* app presents a consent dialog to get user's approval by invoking the system's Package Installer Activity (PIA) before app installation. Either way, the installer goes through four steps to complete the transaction: (1) it gets an installation request; (2) it downloads a new app itself or through the Download Manager (DM) to the SD-Card or internal storage; (3) it invokes the PMS or PIA for installation; (4) the PMS or PIA installs the new app. Note that DM, PMS and PIA are the building blocks Android provides to the app developers; all other design and implementation details of the app installation transaction (AIT) are left in their hands.

This treatment is in line with Android's design philosophy, which fosters diversity with minimal restrictions from the framework end. However, for critical functionalities like app installation, one may question the decision to leave the design and implementation to the 3rd party app developers. If something goes wrong, the consequences could be serious, impacting not only the app itself but the whole system. For example, a vulnerable installer app could be exploited to install a malicious app or even a problematic system app to access sensitive user resources or even gain system privileges.

Ghost installer attacks. Our scrutiny of the app installation transaction (AIT) and popular installers reveals that almost all AITs are vulnerable and can be exploited, and every step of the AIT contains security-critical weaknesses. At the first step of AIT, we found a vulnerability in Amazon AppStore (2nd most popular Android appstore [18]) that allows a malicious app in the same device to command the AppStore to install or uninstall any apps. A more generic threat that affects all appstore apps is the *redirect Intent* attack: when an appstore app receives a request to display an app for installation, a malicious app in the background can change the app being displayed to a different app. Also, a vulnerability we found in Android DM allows a malicious app to damage, redirect, acquire the file or even deny an app installation.

Most interestingly, we found that except Google Play, most apps (including Amazon, Qihoo360, etc) utilized the SD-Card to temporarily hold the APK file to be installed. Although most installers have put effort to secure the SD-Card based installation (e.g., integrity verification of APK files), by defeating them, we show that building such protection is actually non-trivial: we demonstrate the Time of Check to Time of Use (TOCTOU) vulnerability in *all* installers using the SD-Card, in which the malicious app can effectively identify the time window between the integrity check and the installation to

2158-3927/17 $31.00 © 2017 IEEE
DOI 10.1109/DSN.2017.33

replace the APK files. The impacts of the attack are significant, enabling the adversary to leverage the installer apps with to silently install any third-party apps including system apps.

We successfully implemented all the exploits (demos [8]), dubbed *Ghost Installer Attack (GIA)*, and demonstrated their significant impacts. Particularly, by exploiting Digital Turbine Ignite (DT Ignite), an app used by 30+ world's leading carriers (Verizon, AT&T, T-Mobile, Vodafone, Singtel, etc) [12] to push apps to their customers, GIA was shown to be capable of affecting hundreds of millions of users world-wide.

Defense against GIAs. We propose two solutions that address GIAs in SD-Card based installation without requiring the developers to use the internal-storage. One approach modifies the FUSE daemon, a wrapper of raw storage devices, to set the APK files to `read-only` but writable only by its owner as they are downloaded. The other approach only requires an unprivileged app, which detects an installation event, collects the signature of an app's certificate before it can be replaced and later verifies it against the app installed. Further we developed a set of system solutions to address the security risks caused by the redirect Intents. All new protection mechanisms have been evaluated, found to work effectively and efficiently against GIAs, incurring negligible performance impact.

Contributions. The contributions are outlined as follows:

- *Systematic study on App installation*. We conducted the first systematic study on the app installation process on Android and discovered significant security risks never known before. Our research demonstrates that every step of the installation transaction contains security-critical flaws that can be exploited, opening the door to serious security breaches (installing unwanted apps, privilege escalation, etc.) with devastating consequences. The impacts of our findings are significant, affecting majority of installer apps, most Android devices in the market and hundreds of millions of Android users. Most importantly, our study points to the problem in Android's design philosophy, highlighting the need to identify the functionality with system-wide impacts and ensure that it is securely designed and released to the public as a fully-developed service.

- *New protection*. With deep understanding of GIA threat, we present a set of lightweight and effective defense strategies that are non-trivial as we achieve it with minimal modification on Android. Our system-level protection can be built into systems without undermining the strategic decisions (using SD-Card) of appstore app developers. The user-level defense app can protect users even when they are using insecure installer apps.

II. Background

Nuts and bolts. Android apps can be installed in two ways: either through an app with the `INSTALL_PACKAGES` permission or the `PackageInstallerActivity` class. In the former case, an app granted with the permission can silently install new apps by calling the `installPackage` method of the `PackageManager` class. Such an app typically also has the `DELETE_PACKAGES` permission to silently uninstall apps. Due to the critical functionalities of the permissions, their protection-level are `signatureOrSystem`, which is the *highest* that can be assigned to a permission. The latter approach, utilized by less privileged apps, prompts a consent dialogue to users to get approval for installation.

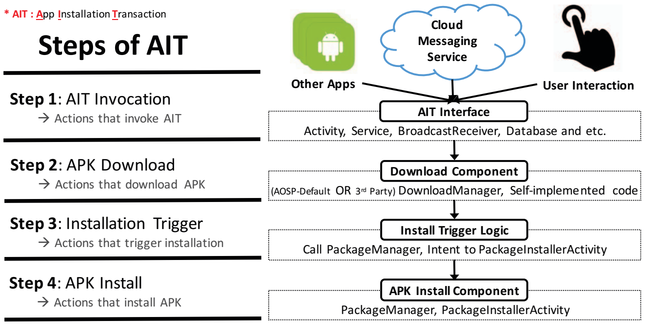
* AIT : App Installation Transaction
Steps of AIT
Step 1: AIT Invocation
→ Actions that invoke AIT
Step 2: APK Download
→ Actions that download APK
Step 3: Installation Trigger
→ Actions that trigger installation
Step 4: APK Install
→ Actions that install APK
Cloud Messaging Service
Other Apps
User Interaction
AIT Interface
Activity, Service, BroadcastReceiver, Database and etc.
Download Component
(AOSP-Default OR 3rd Party) DownloadManager, Self-implemented code
Install Trigger Logic
Call PackageManager, Intent to PackageInstallerActivity
APK Install Component
PackageManager, PackageInstallerActivity

Fig. 1: App Installation Transaction (AIT) steps

Note that protection-level `signatureOrSystem` permissions such as `INSTALL_PACKAGES` and `DELETE_PACKAGES` are only granted to apps that are part of the system image or signed with the phone's *platform key*. Hence, parties capable of pre-installing apps to the system image or controlling the phone's *platform key* are given the privilege to grant those permissions with responsibility. Examples of the parties here include phone manufacturers, carriers, etc. However as shown in Section IV, surprisingly, 10% of the pre-installed apps were granted with the INSTALL_PACKAGES permission. Throughout the paper, we show the serious consequences that can be caused by such permission abuse.

App Installation Transaction. As shown in Figure 1, a typical transaction includes `AIT Invocation`, `APK Download`, `Installation trigger` and `APK Install`. More specifically, an AIT starts with an Intent delivered or a set of user interactions that trigger the whole transaction, which is followed by the download of the target_apk (the APK file to be installed) and its related meta-data, such as a hash of the binary file for verification. This step can be performed through Android Download Manager or also be self-implemented. Once the download is completed, an Intent or an API can be used to launch the installation process, through the PIA or PMS. Finally, the PMS or PIA installs the app, interacting with the user (e.g., consent dialog) when necessary.

As mentioned earlier, AITs are implemented by different app developers, based on the nuts and bolts (e.g., permission, PMS, PIA, etc.) provided by Android. As a result, design and components of each AIT can vary (see Figure 1). Also the detailed execution of AITs are actually quite diverse: e.g., some apps are first installed from appstores and then updated from their developers' sites; some pre-installed apps silently install other apps and others are side-loaded by users. With such diversity, it becomes less clear whether these AITs are indeed implemented in a safe way.

Understanding SD-Card usage of 3rd party appstores. Storage selection in Android is much subtler than it appears to be. To install an app, developers need to temporarily save the app file (.apk) either to the internal storage or the SD-Card. *An obvious option seems to be* the *internal storage* since it is more secure (by default, apps can only access its own folder) while folders in the SD-Card are exposed to any app that has the `WRITE_EXTERNAL_STORAGE` permission. Surprisingly, however, besides Google Play, as shown in Section IV, most 3rd party appstore apps including Amazon appstore and Qihoo360 (a renowned security company with a revenue of 1.8 Billion USD in 2015) choose the SD-Card even if the appstore owners have to put additional protection in place to ensure the integrity during the app install process. Examples of such protection include: 1) elimination of the permission consent dialog to ensure that the APK file cannot be modified when the user is viewing the consent (e.g., the Amazon appstore app installed on Galaxy S6 Verizon), 2) APK file name randomization to prevent the attacker from locating the file and replacing it, 3) hash verification of APK file after download, 4) use of the Android API `installPackageWithVerification` for integrity check, 5) DRM in Amazon appstore apps (Section III) and others. With such effort, the prevalence (97.1% of pre-installed apps, 83.7% of Google Play apps) of these potentially vulnerable installers (Section IV) still becomes a serious concern. In our research, we investigated the rationale behind this insecure storage choice. Apparently, the driving force here is the need to be *compatible with low-end phones*.

Installing apps through internal storage takes *twice the internal storage space* compared to using the SD-Card: 1) space for installing the app and 2) that for temporarily storing the APK file until installation is finished. Due to such space requirement, app installation fails when devices have insufficient internal storage left; low-end devices with small internal space has a higher chance to suffer from such problem. For example, if the Amazon appstore used the internal storage to install `Gabriel-Knight-Sins-Fathers-Anniversary` (1.6GB [2]), the attempt would not succeed on a Galaxy J5 (the 8GB model with only 2.5GB left for third party apps). Actually, the low-end devices with small internal memory (4GB or 8GB) are popular[1] as much as the flagship devices; during the first half of year 2016, 13 million Galaxy J2 (8GB) and 11.8 million Galaxy S7 (flagship device from Samsung) were sold [19]. Also, in Feb 2016, Gartner reported that basic and lower-end devices will account for two-thirds of smartphone sales by 2019 [7]. Most importantly, low-end device users are only left with a *fraction* of internal storage because *system memory* and *pre-installed apps* also takes space.

Under such circumstances, compatibility with low-end devices becomes important to appstores, which are consistently under the competitions from their peers. An exception here is Google Play, which is not only dominating the app market but also is pre-installed on most devices. During our study, we found APKPure, an interesting appstore that became popular (Alexa Ranking as of Aug 14, 2016: Global:2,674, India:895 [1]) by just providing Google Play apps so that users can install them through the SD-Card; APKPure has been recommended to those who are suffering from limited internal storage [15]. This gives us a reason to believe that 3rd party appstores and users may prefer to leverage the SD-Card to increase the chance of successful installations, even though its security risks are well-known.

Understanding SD-Card usage of ordinary developers. Interestingly, most (83.7%, see Section IV) apps from Google Play also use the SD-Card for app installation. While appstore app owners have some protection in place to secure the SD-Card based installation, ordinary app developers may choose the SD-Card due to the simplicity of implementation and the lack of proper secure programming training. To install an APK using internal storage, the access-control permission of the APK's file needs to be set to global readable. Otherwise, `PackageManager` cannot read it. Searching the error caused by this read failure on stackoverflow, multiple answers suggest storing the APK to the SD-Card instead. However, while this will allow `PackageManager` to access the APK, it will also enable any app with the `WRITE_EXTERNAL_STORAGE` permission to tamper with the file. The tendency of developers to apply suggestions from websites without understanding the potential risks has created severe security issues before [33].

TABLE I: Summary of AIT problems

Section	Attack Name	AIT steps [Step No]
3.2	Hijacking Installation	Installation Trigger[3]
3.2	Hijacking Installation	APK Install[4]
3.3	Exploiting DM	APK Download[2]
3.4	Attacking Installer Interfaces	AIT Invocation[1]

III. Ghost Installer Attacks

In this Section, we elaborate our analysis on real-world AIT implementations, which has led to the discovery of unexpected, security-critical vulnerabilities throughout different AIT steps. The consequences are serious: we show that a malicious app, with very limited privilege (discussed later), can install *any* apps, gaining any dangerous level permissions without user's consent and even access to system resources. The problem affects hundreds of device models and hundreds of millions of Android users worldwide (Section IV).

A. Overview

Surprisingly, every AIT step turns out to be vulnerable: AIT invocation (Step 1) is subject to code injection and redirect Intent attacks (Section III-D), APK download (Step 2) is under a threat aiming at a subtle weakness within the AOSP download manager (Section III-C), installation trigger (Step 3) of most popular app stores (e.g., Amazon, Xiaomi, Baidu) and pre-installed apps (e.g., DTIgnite) are all exposed to a TOCTOU attack that swaps the packages to be installed (Section III-B) and APK install (Step 4) is also vulnerable due to the way the PMS and PIA verify target_apk. The rest of the section is presented by following the complexity of the problems: Step 3, 4, 2 and 1 (See Table I).

Adversary model. We consider an adversary that places a malicious app on a user's device, which is a common requirement for Android-based attacks [27, 22, 37, 42]. For some targets (Section III-B), the malicious app requires `WRITE_EXTERNAL_STORAGE` permission, a common permission used by many apps (see Section IV). From Android 4.4, private storage directory on `/sdcard` is supported. However, such directories are not actually private and are writable by any app with `WRITE_EXTERNAL_STORAGE` permission [5]. Note that even with the advent of the runtime permission request model introduced on Android 6.0, granting this permission can be made unnoticeable to the user: the `WRITE_EXTERNAL_STORAGE` and `READ_EXTERNAL_STORAGE` permissions belong to the same permission group, i.e. `STORAGE`; if an app requests a permission in the group with an already granted permission, the requested new permission will be given silently by the system [13]. So, the adversary can request one of the permissions in the STORAGE group for a legitimate purpose and then silently acquire the others.

B. Hijacking Installation (AIT Step 3 and 4)

Most app installers (for app stores) today use the external storage to install apps (new apps or updates). More specifically, the installer initiates the AIT by downloading the target_apk

[1] 30 devices with 4GB and 78 with 8GB are sold at bestbuy.com

and metadata (e.g. the hash of the apk) from its server and storing it to the SD-Card. After verifying the hash of the package, APIs are called or Intents are sent by the installer to invoke the PMS or PIA to install the target_apk. The use of the SD-Card is considered necessary for most apps, including almost all major appstore apps (e.g., Amazon, Baidu, etc.) except manufacturers like Google and Samsung. The manufacturers can always pre-install their apps on the devices they control, while the third-party installers are under peer pressure and need to make use of every bit of storage available to install or update their apps on their customers' devices. This becomes particularly important when the installer is designed to work on the devices with different internal storage sizes, including those with very small space (see Section II).

TOCTOU exploit. External storage is well known to be insecure, readable and writable by the app with READ_EXTERNAL_STORAGE and WRITE_EXTERNAL_STORAGE permissions. However, an attack on the storage to tamper with the installation process is more complicated than it appears to be. Specifically, system app developers and appstore owners are well aware of the security risks and they perform integrity checks to ensure that packages are not improperly modified before installation. A direct attempt to replace the files downloaded will be defeated if it does not happen within the right *time window* (right after the integrity check and before installation). Actually, some installers have already made an effort to minimize such time window. Particularly, Amazon and Xiaomi appstores call the PackageManager.installPackage API immediately after the hash verification is completed. Also, Amazon randomizes the names of the packages downloaded to avoid being identified by an attacker. To safeguard the installation process, Android also provides a hidden API PackageManager.installPackageWithVerification (AIT Step 4) for the PMS to verify the integrity of an app's manifest (AndroidManifest.xml) provided by the installer.

In spite of such protection, our research shows that a TOCTOU attack replacing a target_apk with a malicious one can still be reliably executed. Particularly, any app with the above SD-Card permission can use the FileObserver class to monitor the file access events under a given directory. Even though some appstores randomize the names of the packages to be installed, the directory path turns out to be less convenient to change, given the practice that many appstores pre-download some apps the phone user might want to install in the future on her SD-Card and the difficulty in managing random directories for all these apps on the card. Actually, even when this indeed happens, the attack app can simply keep track of all such directories and find the right targets under the directories according to their types. In our study, we found that the attacker, with nothing but the SD-Card permission, can register with the FileObserver and can be notified with file-access events such as CREATE, ACCESS, CLOSE_WRITE and CLOSE_NOWRITE. As a result, it can catch the window based on how files under the directory are used.

Specifically, our attack app waits until the target_apk is created and its integrity check is done and then substitute a malicious APK for the file. Completion of the file download can be easily known by observing the CLOSE_WRITE event. A bit tricky here is to detect the integrity check. Different appstores may read from the file different number of times, depending on the implementation of the check. The strategy used in our attack is to analyze the target appstore beforehand, figuring out its access pattern. For example, for the Amazon app, as soon as 7 CLOSE_NOWRITE events are observed right after the completion of the download, we know that the check is done and it is time to do the replacement; for Qihoo360, we found that 3 CLOSE_NOWRITE events need to be seen before the attack can be executed. This approach was very effective, enabling us to successfully attack most appstores and the updating process of popular apps.

Note that FileObserver is by no means the only channel for capturing the attack window. Even when it is closed (e.g., requiring new permissions to monitor events), there are other ways to determine the timings for the attack. For example, we can determine whether the target_apk is fully downloaded by looking for the presence of its *end of central directory record* located at the end of the file, and then wait for a short period of time, as measured beforehand on the device of the same model, before start replacing the file (by moving a pre-stored file to the directory). In practice, this simple "wait-and-see" strategy works very well. Note that even when the window is missed and the corrupted file is detected, many appstores and apps re-download the target_apk, a process transparent to the user, which enables the attacker to try again.

Once the target_apk is replaced, only protection left is the manifest verification which is done by PMS during installation. This defense can be circumvented by a malicious app utilizing the original app's manifest, which typically happens in app repackaging. Following we present few attacks on popular appstores and apps.

Attack on DTIgnite (Step 3). DTIgnite is a pre-installed system app used by major carriers (Verizon, T-Mobile, etc.) to push (post-sale) bloatware to their customers (Section IV). For this purpose, it is capable of silently installing apps on Android devices. The APKs of such apps are stored on the external storage (/sdcard/DTIgnite) by the DM and verified against their hash values before installation. In our research, we successfully attacked DTIgnite using our attack app (with only SD-Card permission) through both the FileObserver monitoring and the "wait-and-see" strategy (by waiting for 2 seconds after download) on Galaxy S6 Edge (Verizon). Given that the system app is used by over 20 carriers, the attack affects hundreds of millions of Android users (Section IV).

Attacks on Appstore apps (Step 3). Amazon, Xiaomi and Baidu are popular Android application stores, which have been pre-installed on a large number of devices. For example, Amazon app (com.amazon.venezia) routinely appears on Android phones with Verizon and USCellular (see Section IV). All these appstores utilize the same approach as DTIgnite to install an app, except that they immediately activate the PMS once the download and the integrity check are complete. However, the protection does not work at all in the presence of the FileObserver monitoring: our attacker successfully caught the attack window after observing 7 CLOSE_NOWRITE events for Amazon and 1 for Xiaomi and 2 for Baidu. Alternatively, we found that Amazon and Baidu can be attacked in the same way as DTIgnite, with the only difference that the replacement needs to be done 500ms after the download is completed. For Xiaomi appstore, its download completion can be easily identified from the installer's behavior that changes the temporary name of the target_apk to its official name.

Appstore apps can also be side-loaded by users as a non-system app. In this case, unless the appstore app is signed by the platform key, the app installation goes through the PIA, which pops up a user consent dialogue during installation with the target_apk's package name and package icon. This protection is defeated in our study by embedding within the malicious APK (for replacing the target_apk) the original app's name and icon. Note that the target_apk replacement time window can be detected using the same `FileObserver` approach as above. Like installation of a new app, updating existing apps can also be attacked in the same way.

Attack on new Amazon appstore (Step 4). Of particular interest here is the upgrade of the Amazon appstore from May 7th, 2015 to version−17.0000.893.3C_64 7000010. This version includes protection under the `installPackageWith Verification` API (AIT Step 4), which takes the path of target_apk and the checksum of target_apk's manifest (AndroidManifest.xml) as a parameter and verifies the checksum before installation within the PMS, and Digital Right Management (DRM) features that self-check whether it has been tampered with. All such protection is defeated in our research through repackaging an Amazon app with attack code, removing its DRM code but keeping its manifest. Note that this version has two hash verification protection in place, one done by Amazon appstore itself and the other by the PMS.

Attack on PIA (Step 4). To prevent the target_apk from being replaced while the permission consent dialogue is displayed to the user, the PIA records the hash checksum of the target_apk's manifest before the consent dialogue and verifies it prior to installation. However, as we can see, this measure cannot defend against the attack on Step 3, since the windows between the integrity check and the follow-up step can still be *reliably* captured. Actually, the protection does not even work on Step 4, which it was designed to secure: what the adversary can do is simply substituting a malicious APK for the original one using the *same* manifest (e.g., a Phishing version of a bank app) to completely defeat the defense. This vulnerability demonstrates the complexity of AIT and the fact that designing protection without fully understanding the root cause of the problem cannot succeed. In Section V, we present a new solution that is practical and effective.

Privilege escalation. Through unauthorized app installation, our research further shows that the adversary can acquire higher privileges, including system level privileges. Specifically, on Android, any apps signed with the manufacturer's platform key are given system permissions at the signature protection level. By leveraging the way the platform key is used in Android, we found that the adversary can silently install system apps onto the user's device, as elaborated below.

One way to gain a higher privilege is to deliberately install a vulnerable system app signed with the same platform key as the victim's device and then exploit it after installation. Since Android does not allow the existence of two apps with the same package name, this can only be done in the absence of the patched version of the vulnerable app, which is found to be feasible in our research: due to the fragmentation of Android, numerous devices of the same vendor are available on the market, each with a different set of system apps. Most importantly, those apps and many others in the Play store are all signed with a *single platform key*, which gives the adversary a lot of attack opportunities. In our research, we ran our malware that stealthily installed vulnerable Teamviewer and later exploited it using the techniques reported by Check Point [25] to gain system privileges.

Another path for privilege escalation is to exploit *Hare (Hanging Attribute Reference)* permission. Hare permission is a permission that is used to guard user resources but has not been defined by any app on a device. The problem has been found on popular Android devices by the prior research [21]. Using GIAs an attacker can *create* a Hare situation by deliberately installing a system app that uses a permission undefined by any legitimate app on the device, which enables the malware to grab the permission by defining it. As a result, the attacker can utilize the permission to access the resource it is not supposed to touch. In our research, we successfully implemented this attack through installing a Hare-creating system app (S-Voice and Link) on Galaxy Note 3. The attack enables the malicious app to hijack `com.vlingo.midas.contacts.permission.READ`, `com.vlingo.midas.contacts.permission.WRITE` permission and use them to steal the user's contacts. The scope and impact of the problem are discussed in Section IV.

C. Exploiting DM (AIT Step 2)

As mentioned earlier, at the center of the APK download step is a download manager, which is typically the default AOSP DM. To use the DM, an app provides the URL of the target_apk and the destination file path as inputs, and receives an ID from the DM to later retrieve the file and related information or delete the file through the DM. During this process, the DM enforces a set of security policies, binding the requesting app's package name to the ID and also to the file path the app is authorized to access, further ensuring that the path points to either the `/sdcard` or its cache folder. However, we found in our research that the file path can be symbolic, which brings in another TOCTOU risk. Note that from Android's perspective, symbolic paths have to be supported, since they are extensively used, including the SD-Card directory `/sdcard`. The problem is that this approach also gives the adversary an opportunity to change the link, pointing to a different directory after the check, if the access control has not been well managed.

The attack. Indeed, our study shows that such an attack can actually succeed on the Android's DM. Specifically, we show that our attack app was able to escalate its privilege by first requesting the DM to download a file to a symbolic path A pointing to an authorized location (e.g., somewhere on the SD-Card) and once the download is completed (indicating that the security check on A is done) re-mapping A to a different physical path B where the app does not have the right to access but the DM has. As a result, the attacker can acquire the DM's privilege to retrieve other apps' files or even delete them.

We verified the vulnerability in Android 4.4 and 6.0. The consequence of our attack is serious: our attack app was able to delete any files the DM is allowed to remove, including the DM's database, which caused a denial-of-service attack on Google Play; further, the attacker acquired any files the DM had access to, including files the DM downloaded and even the DM's database. Interestingly in Android 6.0, the DM actually checks the physical path of a symbolic link right before processing an access request. However, there still exists a gap between the check and the actual processing of the request, which can be exploited to redirect the link to another path. In our study, we ran a process that continuously changed the mapping of the links, trying to capture the window. We successfully attacked the DM as shown in our demo [8]. This problem is serious because it enables a malicious app with no permission to tamper with the files even in the internal storage (considered to be secure).

The information leaked through this channel not only informs the adversary an ongoing app installation operation, which can be utilized to launch the hijacking attack mentioned above (Section III-B), but also leads to the disclosure of other sensitive information, including Android's secret URL tokens that might be used to gain unauthorized access to the Play store, though its detailed use is kept secret by Google. Our findings reported to Google are rated as a **high-severity** issue. The issue has been fixed with our help.

D. Attacking Installer Interface (AIT Step 1)

At the AIT invocation step, an installer is activated by an Intent that initiates the whole installation process. Our research shows that even this simple boot-strapping step is full of security risks, vulnerable to a *redirect Intent* attack on the installer's user interface (UI) and *code injection* through its Intent processing interface. Specifically, an installer can be invoked by another app (the AIT initiator) through an Intent and its UI can be directed by the Intent to the activity displaying the app recommended by the initiator. The trouble, as found in our research, is that a subsequent Intent sent by a *background app* can cause a new UI change before the former can be perceived by the user. By exploiting this weakness, the background malware can cheat the user into installing a malicious app from the appstore. It is important to note that this attack is different from the well-known UI Phishing attacks [28, 37] in which malware launches its *own fake activity* to cover the foreground app's UI; such an old trick *cannot* cause the user to install a wrong app, since the malware's *own fake activity* cannot trigger any installation transaction without the proper permission. On the other hand, Redirect Intent attack leverages the victim app's (Google Play in our case) activity and does not require any fake activity. Because of the difference, previously proposed protection such as [22] does not work against our attack. Actually, the weakness behind the new attack is fundamental, coming from the design of Android's `ActivityManagerService`, which allows the background app to redirect the installer's UI within a very short time frame (see our demo [8]) without providing the Intent recipient (the installer here) the origin of the redirection Intent. As a result, whenever the user is redirected from an app she trusts, very likely she will also trust the app it recommends, without knowing that the UI has actually been stealthily changed to display a different app.

Further, we found that the implementations of some popular appstores' Intent processing interfaces have not been well thought out, missing proper authentication about the sender of the Intent. Note that Android does not provide a mechanism to let the Intent receiver find out the sender's identity. The consequence is that they could blindly act on the commands included in the Intent, installing malicious apps or deleting legitimate apps on behalf of the adversary. Following we describe the exploits we performed on popular appstore apps.

Redirect Intent attack. In our work, all appstore apps were vulnerable to the redirect Intent attack. A prominent example is Facebook app's invocation of Google Play for installing Facebook Messenger. Such an invocation, goes through `ActivityManagerService` that utilizes the parameters within the Intent to direct Google Play to show the right UI. The key to a successful attack is the timing: the malware in the background needs to know exactly when Facebook app sends the Intent to Google Play. For this purpose, we utilized a side channel: the malware continuously polls `/proc/<pid>/oom_adj` (zero when the app is in the foreground) to monitor the victim app (Facebook in this case). As soon as Facebook app leaves the foreground and Google Play takes its place, the malware also sends an Intent to Google Play, asking it to display a Facebook Messenger looking like app. This transition is invisible to the user, as demonstrated in our demo [8]. Note that this attack requires the adversary to have knowledge of the legitimate app that is requested to be installed (the Facebook Messenger app in this case). This is needed for the adversary to prepare a repackaged version or similarly looking version on Google Play. As we show in Section IV, a large number of apps redirect to predictable legitimate apps and thus are vulnerable to the redirect Intent phishing attack.

Command injection. We found Amazon appstore app to be vulnerable to command injection attacks because its public activity com.amazon.venezia.Venezia receives other apps' Intent for referring users to other apps. Specifically, the problem is caused by Amazon's `MainActivity`, whose WebView component supports a Javascript-Java bridge which enables Javascript services on Amazon cloud to run Java services on a mobile device to perform app installation/uninstallation. The problem is that the activity fails to authenticate the origin of the Intent and check whether it includes Javascript code. As a result, it blindly executes the commands activated by the script. In our research, we ran a background app that sends an Intent using `single top` mode, which ensures that Amazon app's activity would not be destroyed and recreated. As Amazon app received the Intent, it ran the script within the message and silently installed/uninstalled any app. We found that through this approach, a malware can actually invoke **any private services** of Amazon app, substantially escalating its privilege.

Further we discovered a security-critical problem in Xiaomi appstore. The appstore utilizes a broadcast receiver to get messages pushed from the cloud, which can command it to install or uninstall apps. However, the appstore never authenticates the messages it receives. As a result, we were able to send an Intent to the receiver, causing the appstore to `silently` install an app we chose without being noticed by the user. We broadcasted an Intent with a forged payload destined to be captured by Xiaomi Appstore's broadcast receiver. The forged payload[2], included the app id and package name of the malicious app stored on Xiaomi appstore and due to the absence of authentication the malicious app was silently installed to the victim's device.

[2] `{"jsonContent":"{"type":"app", "appId":"xxx", "packageName":"xxx"}"}`

IV. Measurement

In this section, we report a large scale measurement study to investigate the scope and magnitude of GIA threats.

A. *Methodology and Data*

Factory image and app collection. To understand the potential security risks, which come from the prevalence of `INSTALL_PACKAGES` within vulnerable pre-installed apps and the management of platform keys, we crawled factory images from Samsung-updates [16], official Xiaomi [20] and Huawei [11]. We selected the images from Android 4.0.3 to 5.1, which covered 95.7% (as of Nov 2, 2015) of the devices that visited Google Play [3]. Among those, we downloaded **1,239** Samsung images for **849** different device models, **382** Xiaomi images for **149** devices and **234** Huawei images for **135** devices. Such images include **231** distinct regional codes, covering **79** countries and various carriers across the world. From those factory images, we extracted **206,674** distinct (based on md5) pre-installed apps, together with their signatures (under their `/META-INF`) and the certificates of the platform keys within the images (from framework resources). In addition, we analyzed **top 13,500** free apps (with **12,750** unique ones) from Google Play (top 500 free apps from 27 categories) to understand the impacts of the GIA threats. Furthermore, to find out how many apps in appstores are signed by vendors' platform keys, we extracted signatures from **1.2 million** apps downloaded from 33 appstores, including 400,000 Google Play apps.

Finding potentially vulnerable installers. Finding potentially vulnerable installers turns out to be more complicated than it appears to be. A straight-forward solution, using an information flow analysis to identify the use of insecure external resources for installation[3], does not work well in practice, due to the complexity of the analysis. Specifically, we tried to build a static tool upon Flowdroid [23] to find out whether an installer is actually using the SD-Card to install apps. However, the attempt failed on many apps due to the limitations of Flowdroid. Among the 43 apps we tested, 14% was stopped by incomplete Control-Flow Graphs, another 14% failed because tainted data propagated through the `handlemessage(message)` API that cannot be tracked with the call graph provided by Flowdroid and 42% were disrupted by the bugs in Flowdroid. Moreover, to cover different implementation details of installers, we would need to consider all channels each AIT modules can use to communicate with others (Java reflection, Handler and etc.), which turns out to be too complicated and unreliable for a measurement study.

To address the issue, we leveraged a unique observation discovered in our research: to install an APK using the internal storage (the secure option), *the installer needs to make the APK `global-readable`*; otherwise `PackageManagerService` will not have the permission to access the APK to install it. Based on the observation, we built a simple yet effective tool that quickly and accurately identifies vulnerable installers. Specifically, our tool first finds the APKs including the installation API calls and the `WRITE_EXTERNAL_STORAGE` permission. For this purpose, it uses Apktool [6] to decompile APKs to search for `"application/vnd.android.package-archive"`, the installation API code. Then, on those determined to be installers, the tool checks whether they contain APIs for setting a target_apk to `global-readable`. This is done by running Soot [17] to transform `dex` code to `jimple` for retrieving related APIs, including `openFileOutput(filename,Context.MODE_WORLD _READABLE)`, `setReadable()`, execution of `chmod XXX /FilePath`, `setPosixFilePermissions()` etc. Once located, the input variables of these APIs are further analyzed through def-use-chain to confirm that indeed right parameters are there (e.g., `MODE_WORLD_READABLE`). Based on the results of this analysis, our tool automatically classifies apps into three categories: (1) potentially vulnerable apps, which call installation APIs and operate on `/sdcard` but do not set the target_apk `global-readable`; (2) potentially secure apps, which do not use `/sdcard` and also set the target_apk `global-readable`; (3) unknown apps, all other installers.

TABLE II: Potentially vulnerable GooglePlay apps due to SD-Card usage

Type	SD-Card (Potentially vulnerable apps)	Internal Storage (Potentially secure apps)
Excluding Unknown Apps	779/931 (83.7%)	152/931 (16.3%)
Including Unknown Apps	779/1493 (52.2%)	152/1493 (10.2%)

TABLE III: Potentially vulnerable pre-installed apps due to SD-Card usage

Type	SD-Card (Potentially vulnerable apps)	Internal Storage (Potentially secure apps)
Excluding Unknown Apps	102/105 (97.1%)	3/105 (2.86%)
Including Unknown Apps	102/238 (42.9%)	3/238 (1.26%)

Running this simple tool on 12,750 apps from Google Play, 1,493 had installation API calls. Among the 1,493 apps, 779 were classified as first category, 152 as second category and 562 as last category. We also analyzed 12,050 pre-installed apps extracted from the selected 60 images (20 images from Samsung, Xiaomi and Huawei which covers diverse device models and versions). Removing the duplicate apps (based on package name) left us 1613 apps; different devices from the same manufacturer contains similar set of apps. Among them 238 had installation API calls, 102 were classified as first category, 3 as second category and 133 as last category. By randomly sampling and manually analyzing (reading smali) 20 apps from each category, we confirmed that all apps in "potentially vulnerable" category are vulnerable without false positives. Similarly, Apps in "potentially secure" category were secure without false negatives. Therefore, our measurement based on those in the first category is conservative, including only a subset of vulnerable installers.

To understand the impact of the redirect Intent threat (Section III-D), we identified apps that redirect users to Google Play by using either the URL[4] or scheme[5]. This was done by inspecting the smali code of the apps for the URL or the scheme. Note that this approach could miss dynamically constructed links, which again makes our findings conservative (the real impact can be even more significant).

B. *Results*

Pervasiveness of the SD-Card usage. Our study shows that the apps vulnerable to the installation hijacking attack

[3] Note that simply checking the use of SD-Card is insufficient, since installers may use both internal and external storage, in a safe way.

[4] "http://play.google.com/store/apps/details?id="

[5] "market://details?id=", "https://market.android.com/details?id="

TABLE IV: Number of fixed url or redirection scheme

# of hardcoded url or scheme	1	<=2	<= 4	<= 8
# apps	5.7% (723/12750)	11% (1405/12750)	16.4% (2090/12750)	18.3% (2337/12750)

12750: Top 500 apps from 27 categories of Google Play. Duplicates removed.

(Section III-B) are indeed pervasive, as illustrated in Table II. Among the top 12,750 Google Play apps, 1,493 contain installation related APIs, 83.7% of them were found to install apps through the SD-Card and only 16.3% use the internal storage. Even when we consider all the unknown apps to be secure (which are certainly not), still 52.2% of the installer apps appear vulnerable due to the way they use the SD-Card for installation. Similarly, as shown in Table III, among 12,050 pre-installed apps, 238 contain installation related APIs, 97.1% of them use the SD-Card and only 2.86% (3 apps[6]) use the internal storage. Furthermore, we found that 8,721 out of 12,750 apps from Google Play and 5,864 out of 12,050 pre-installed apps require the `WRITE_EXTERNAL_STORAGE` permission, the sufficient condition for hijacking installations, which indicates that the bar for the attack is rather low.

Impact of vulnerable installers. Table V presents the impacts of the vulnerable installer apps we discovered. As we see, those apps impact **hundreds of millions of users globally**. We further tested popular appstore apps (Baidu, Tencent, Qihoo360, SlideMe) and found that all of them are vulnerable.

To understand the attack surface, we measured the number of system apps with the `INSTALL_PACKAGES` permission. Table VI presents the average number of system apps and the ratio of them with the `INSTALL_PACKAGES` permission per vendor. We see that nearly 10% of the system apps have such privilege. Particularly, found in our study, the number of the pre-installed apps with the permission has **doubled** in the recent three years. Also, more recent flagship models such as Galaxy 6 Edge Plus, Galaxy S6 from T-Mobile, Sprint, US Cellular, Verizon, SK Telecom etc. have a tendency to include more privileged apps (25-31) with the permission.

Usage of platform key. Surprisingly, from the 206,674 pre-installed apps, we found that all three vendors (Samsung, Huawei, Xiaomi) were using only **one** *platform key* to sign all the device models they released. Each device of the vendors have on average 142/68/84 (Samsung/Huawei/Xiaomi) apps respectively and 884/301/216 apps in total signed by their corresponding platform key. Such signed apps are also distributed through appstores. From the signatures of 1.2 million apps we collected, 61/125/30 apps are signed with the key of Samsung, Huawei and Xiaomi. The majority of them are MDM (Mobile Device Management), remote support, VPN and backup apps. Among them is teamviewer, a known vulnerable app [25].

Privilege escalation. To study the significance of privilege escalation through *Hare* generation (Section III-B), we extracted the apps using the permissions that they themselves fail to define from 10 Samsung images (version 4.4.4 to 5.1.1). Note that these apps can still be secure if the permissions are defined by authorized parties on the same device. On these images alone, we found 178 such apps. The permissions used in these apps were then searched across other 1,181 images, which led to the discovery of 27,763 unique vulnerable cases (where a system app using a permission can be installed on a factory image on which none defines that permission). On average, each of these images has 23.5 vulnerable cases. As a result, a malicious app on the device running such a vulnerable image can install the Hare creating system app (signed with the platform key) and define the missing permission to acquire the resource the permission protects (such as user contacts).

Apps invoking Google Play. We further studied the apps that redirect users to Google Play for installing new apps. In total, 84.7% of the top 12,750 apps on the Play store are redirecting users with the fixed URL or scheme. As shown in Table IV, among them, 723 contain just a single hard-coded URL or scheme which makes them the easy and realistic targets for the redirect Intent attack (Section III-D); there is no confusion about which apps their redirections will lead to. (Impactful redirection examples: Facebook→Facebook Messenger, Power amp→Paid app, Tiny Flashlight + LED→Plugin app)

V. Fighting GIAs

A. Understanding the Problems

Some vulnerabilities in Section III have simple solutions, while other harder ones require new techniques (Section V-B).

Flaws with quick fixes. Below we discuss the cause of each problem with the solution we suggest.

• *Amazon and Xiaomi appstore*. The problem comes from unauthorized execution of the malicious payload (Javascript) within the Intent the appstore receives. The script then controls sensitive private API that should not be exposed to the public. The problem can be addressed by input sanitization, filtering out malicious script code, and also limiting the capability of the JSJAVA bridge. We have reported the issue to Amazon and helped them to fix it. Meanwhile, Xiaomi appstore is vulnerable because it exposes its `BroadcastReceiver`, which can be fixed by protecting the receiver with a permission.

• *AOSP DM*. The fix is to ensure that the DM always checks where a symbolic link points to whenever it has been used to access resources. This problem, classified as a **high severity** issue by Google, has been solved according to our report.

• *Verification API* . As mentioned earlier (Section III-B), Amazon appstore and the PIA are using `installPackageWithVerification` to verify the integrity of an app's manifest file before installing an app. This approach is insufficient, since it can be defeated by a malware using the same manifest file as the app being replaced. A better solution is to save the signature of the app once it is downloaded and then verify the signature during the app's installation. This can ensure the integrity of the app during the APK install stage (AIT Step 4), as the original API was designed for (Section VIII).

Harder problems. The *installation hijacking* risk (Section III-B) is caused by the use of the shared external storage. The problem cannot be easily fixed by asking the app and appstore developers to move the whole installation process to the internal storage, as they need to make full use of the storage to put their apps on the user's device (Section III-B). Also a large number of installer apps including various 3rd party appstores in the wild still use the SD-Card (see Section IV). Considering the openness of Android and the fact that they even come from different countries with different policies, it becomes unrealistic to expect them to shortly use the internal storage. To provide Android users immediate protection, we

[6] 3 secure apps: com.miui.tsmclientj, com.huawei.remoteassistant, com.samsung.android.spay (Samsung Pay)

TABLE V: Impact of vulnerable pre-installed apps with INSTALL_PACKAGES permission.

Vulnerable app	Affected devices	Affected carriers	Affected vendors
Amazon appstore	Android devices from Verizon and US Cellular are affected. Samsung devices from Verizon are affected; e.g., Galaxy S4, S5, S6, S6 edge, Note 3, Note 4.	Verizon, US Cellular.	Samsung, LG, HTC, Motorola and more.
DTIgnite	200+ distinct device models. Devices shipped through the affected carriers are impacted. (50+ million apps are already pushed to users device.) [12]	30+ carriers *worldwide* including Verizon, T-Mobile, AT&T and Vodafone.	Vendors that release devices through affected carriers.
Xiaomi appstore	All Xiaomi devices	Carriers that release Xiaomi device - China Mobile, China Telecom, China Unicom.	Xiaomi.
Huawei appstore	All Huawei devices	Carriers that release Huawei device - China Mobile, China Telecom, China Unicom.	Huawei.
SprintZone	Android devices released from Sprint. note that we manually verified it from smali code. We were not able to actually test the attack.	Sprint.	Vendors that release devices through Sprint.

TABLE VI: Average number of system privileged apps and the ratio of the apps with INSTALL_PACKAGES permission.

	Samsung	Huawei	Xiaomi
Avg # of apps with INSTALL_PACKAGES	17.7/206 (8.45%)	9.4/91.4 (10.32%)	11.4/95.4 (11.87%)

developed a suite new techniques that enable secure use of the external storage for app installation (Section V-B).

Phishing through the *redirect Intent attack* (Section III-D) is also an intricate problem fundamentally caused by the design of Android, which enables a background app to send an Intent to a foreground app, forcing it to change its activity (UI) before it can display its current UI triggered by the preceding Intent from a different app. In our research, we propose an enhancement to Android, which enables the recipient of an Intent to figure out the origin of the Intent, and a novel mechanism to detect malicious Intents.

B. User-level Protection of AIT

We developed a user-level app, called *DAPP*, to defend against the installation hijacking threat. This approach does not change the operating system (OS) or the vulnerable app, but provides the protection through a third-party app that can be distributed through Google Play, which we plan to do. The idea behind *DAPP* is simple: as soon as an APK is downloaded, our app grabs its signature and compares the signature against that of the package once it has been installed by the PMS to detect the replacement attack. Further, *DAPP* can identify any file access operations on the SD-Card that may lead to the compromise of target_apk before the completion of its installation. The app is activated through the startForeground API, leaving a notification in the Android Notification Center. This protects it from being terminated by a malicious app with the KILL_BACKGROUND_PROCESSES permission.

Covering the attack window. At the center of *DAPP* is a situation-awareness module that captures file-access events on the SD-Card. This is done through the FileObserver, which reports the completion of target_apk download using the CLOSE_WRITE event. Note that attacker, also using this event (Section III-B), needs to wait until the checksum verification completes, whereas *DAPP* grabs the signature as soon as the APK is downloaded. The completion of the APK installation is found from the PACKAGE_INSTALL and PACKAGE_ADDED Intent broadcasted by the OS.

Finding race conditions. Also any attempt to replace target_apk will be announced by FileObserver and therefore discovered by our app, which protects the installers that do not check the integrity of target_apk after the download (note that leading installers like DTIgnite, Amazon, Xiaomi and etc. perform integrity check).

Specifically, the attempt to move a file to replace target_apk is exposed by the MOVED_TO event. Deleting the APK and copying the replacement here can be detected from the DELETE (which happens immediately after the download) and CLOSE_WRITE events. Even a more subtle trick, opening the target_apk and gradually modifying the content (imitating the download process) triggers the OPEN and CLOSE_WRITE events. *DAPP* considers *any* CLOSE_WRITE that happens *shortly* after target_apk download completion to be suspicious.

C. System Level Protection

In addition to the user-level approach, we also developed a system-level solution for the GIA risk, which addresses the root causes. Our approach includes changes to the *FUSE daemon* to prevent installation hijacking and IntentFirewall to defend against redirect Intent attacks and support Intent origin identification. We evaluate our approach in Section VI.

Guarding SD-Card with the FUSE daemon. In Android, SD-Card is wrapped in a *FUSE daemon*, which uses FUSE (Filesystem in Userspace) to enforce external storage related access control policy and permission. In our research, we modified the daemon to change the Linux discretionary access control (DAC) scheme to protect an APK in an external storage. Our approach makes an APK read-only but writable only by its owner, that is, the app that requests the download (e.g., appstore app). We altered the derive_permissions_locked method and made it set an APK file's permission to 640 (rw- r– –) as soon as an APK is created. This prevents an unauthorized app from overwriting an APK before installation, as described in Section III-B. Interestingly, changing the file permission failed, since Android allows any app with SD-Card permission to write on the external storage, regardless the file permission (DAC) set to the file. To address this issue, we modified the check_caller_access_to_name method to enforce our policy, which protects APKs. This ensures that all non-system apps, except the owner, cannot alter the file, even with the WRITE_EXTERNAL_STORAGE permission. The access setting of the APK is kept after it is installed, in case the APK needs to be re-installed later. In the meantime, the protected file can always be written by a system process, which allows, for example, the user to delete the file to release the space through Android system settings.

Also, an unauthorized app may attempt to bypass such protection by altering (move, delete or rename) the entire path which includes the target_apk. To prevent such attacks, we maintain a list, called *APK list*, to keep track of 1) the owner (UID) of each APK and 2) the file path of all APKs on the SD-Card. Before any path alteration requests are processed, we look up the list and revoke the request if the requested path contains any APKs that are not owned by the requester UID. This is done by modifying the handle_rename method which is part of the FUSE daemon as well.

Redirect Intent attack detection. To detect the redirect Intent attack, we modified the IntentFirewall class of

TABLE VII: Effectiveness & Complexity

Strategy	Tackled Attack	AIT Step	LOC
User-level app (*DAPP*)	Installation Hijacking	3,4	127
FUSE DAC scheme	Installation Hijacking	3,4	156
Intent Detection scheme	Redirect Intent	1	61
Intent origin scheme	Redirect Intent	1	82

the Android framework to add a new class `intentRecord` (IR) for keeping track of an Intent's recipient package name, delivery time and the caller's Linux User ID (UID). Within the `IntentFirewall`, our code creates an IR record for each Intent sent through the `startActivity` API and keeps it in a hash map using its recipient's package name as the key (so only the last Intent received by the package is preserved). For each new record created, we first retrieve from the hash map the IR for the last Intent going to the same app. If the time interval between these two consecutive Intents is found below a threshold (1 second in our implementation), our approach reports the event to the user as a possible attack. To avoid false positives, we do not raise an alarm for such an Intent pair if (1) they all come from the same app, or (2) they are sent and received by the same app or (3) the sender is a system app or service. Further, since our detection mechanism focuses on the Intents from the `startActivity` API, which is typically used to respond to the user-triggered event (such as a click), we do not expect any benign app to send such an Intent within 1 second after another event also generated by the user's activity. Note that apps belonging to the same author (i.e., shared UID) would not be affected by our detection approach since we record the intent recipient's package name and caller's UID.

Intent-origin identification. As mentioned earlier, the fundamental cause of the redirect Intent threat is the lack of origin information for an Intent received. Otherwise, an Intent recipient, e.g., an appstore, can display the name of the sender to the user to get her confirmation. To address this issue, we enhanced Android to support the delivery of the Intent origin information to the recipient. To this end, we modified `Intent.java` and `IntentFirewall.java` to add a field `mIntentOrigin` that keeps the package name of the sender within the Intent class, together with a pair of new methods `getIntentOrigin` and `setIntentOrigin` (as hidden API) to manage the field. When an Intent passes through the `IntentFirewall`, the modified `checkIntent` function calls `setIntentOrigin` to put the sender information in `mIntentOrigin`, which can be inspected by the recipient by calling `getIntentOrigin`.

VI. Evaluation

A. *Effectiveness and Complexity*

Table VII lists the lines of code (LOC) for *DAPP* and the modifications on the FUSE daemon for mitigating hijacking installations, detecting redirect Intents and identifying Intent origins. As we see, our defense mechanisms are lightweight and could be easily adopted. Moreover, *DAPP* is a regular Android app and will be uploaded to Google Play. Our system-level defenses just include 61-156 LOC of Java and C code.

To understand the effectiveness of our protection mechanisms, we tested them against the attacks described in Section III. As depicted in Table VII, *DAPP* and our FUSE patches successfully thwarted the installation hijacking attacks and our Intent detection scheme successfully captured malicious Intents. Further, with all the protection mechanisms on a Nexus 5, for 45 days we installed 924 new apps and used it for daily operations (email, web browsing, social networking, etc.). During this period, no false alarms were reported and none of the legitimate operations were disrupted by our mechanisms.

TABLE VIII: FUSE DAC scheme performance

Unit:ns	Write (org DAC)	Write (mod DAC)	*%	Read (org DAC)	Read (mod DAC)	*%
Average	1772.06	1768.44	99.80%	750.53	765.7	102.02%

*%: Percentage mod DAC compared to org DAC.

TABLE IX: Intent Detection Scheme Performance

Unit:ns	*Total Time	Our Logic	**Percentage
Average	4804339.08	175247.52	0.30%

*Total Time: Time taken from sending intent to receiving intent.
**Percentage: Average of percentage value from all test cases.

B. *Performance*

We further evaluated the performance of our techniques. Specifically, to find out the delay incurred by our detection and access control protection, we utilized the `SystemClock.elapsedRealtimeNanos()` API, which returns the time since boot, including time spent in sleep. The evaluation was conducted on a Nexus 5 device running Android 5.1.

Overhead of *DAPP*. We measured the CPU and RAM usage of *DAPP* during app installation and when the system was idle. Using OS Monitor app [9], *DAPP* was found to work efficiently, on average using only 0.1%-0.7% of CPU and 6.3MB of RAM during installation and 0% of CPU and 6.2MB of RAM on idle. Note that when *DAPP* extracts signatures from a target_apk, because the target_apk is read into memory, *DAPP*'s CPU and RAM usage rose to 1.5%-45% and 10.7MB-76.2MB, but for only 214.7ms in average. Using GSam Battery Monitor [10], we measured the battery consumption by installing 21 apps in 1 hour: with the total consumption hovering around 20% during the experiment, *DAPP* was responsible for only 0.08%. On idle, *DAPP* consumed nearly 0% from total 11% in 1 hour. With this level of power consumption, one can use the device downloading 21 apps within an hour and use the device throughout the day, with only 0.08% of battery being spent on *DAPP*, which is indeed negligible.

Security enhanced DAC. We evaluated the performance of the modified DAC scheme, in terms of the time it takes to perform a write and a read operation on a protected file. For this purpose, we built an app that creates a file and writes 1 MB to it. We repeated it 100 times. Similarly, the app also read a file with 1 MB repeatedly, 100 times. Table VIII reports the average time for each operation on the modified Android 5.1 vs. the original one. As we can see from the table, the overhead of the modified FUSE was so small that it could not even be measured: our implementation on average ran even slightly faster (1%) than the AOSP for the write operation, due to the variations of the execution time.

Redirect Intent detection. To find out the overhead caused by our Intent detection scheme, we built an app that sends an Intent which starts an activity within the other, and recorded the time it takes for the Intent to be delivered (from the moment the sender calls `startActivity` till the recipient gets the Intent but before it displays the view). We compare the time delay caused by our logic to the total time taken to deliver the Intent. We repeated the test 50 times. Table IX shows the delay caused by our Intent inspection logic within the

[1] To replace the prior Intent's resulting screen, a malicious app needs to send an Intent in 200ms-500ms after the legitimate Intent.

TABLE X: Intent Origin Scheme Performance

Unit:ns	*Total Time	Our Logic	**Percentage
Average	64881655.14	828131.06	1.67%

*Total Time: Time taken from sending intent to receiving intent.
**Percentage: Average of percentage value from all test cases.

modified `IntentFirewall.checkIntent()`. Again, we cannot observe any statistically significant delay caused by our logic, indicating that the overhead is negligible.

Intent origin. To evaluate the performance of the Intent origin scheme we conducted the above experiment (on the Intent detection scheme) again, with our origin mechanism on the modified Android 5.1. The result presented in Table X, again, shows that the impact of our additional code is unnoticeable.

VII. Discussion

Development of Security-critical functionalities. In this work, we systematically analyzed the entire AIT, presenting the diversity of the AIT designs (due to customization) and the problems that exist in each step in detail. The presence of these vulnerabilities points to the challenges in making AIT right in practice and most importantly, questions the fundamental design mentality of leaving such a security-critical functionality open to customizations. We suggest that at least the security-critical functionalities should be taken care of by the OS and the current way of leaving them in the hands of individual app developers can be risky.

Suggestions for developers. In real world, various designs of AIT exist and problems can occur in any stage of it. Thus, problems we discussed cannot be solved with one simple solution. Most importantly, despite the criticalness of AIT, the openness of Android is allowing any developer to develop their own installer app without further guidance. To improve such situation, below we provide the key points which will help developers to build a secure installer app.

- *Suggestion 1. Only use the SD-Card if internal storage space is insufficient.* Developers should always use the internal storage space to install a target_apk when there is enough space. If not, we recommend developers to use the SD-Card with the defense techniques (using FileObserver events) we elaborate in Section V. As discussed in Section II, in certain circumstances SD-Card may be preferred; small internal memory, big apps.

- *Suggestion 2. Verify hash of the target_apk in a secure storage.* Developers should verify the hash of the target_apk in a secure storage (internal storage or SD-Card with our defense in place) right before installation to make sure the file has not been tampered with. This is the last line of protection that can prevent replaced apps from being installed.

- *Suggestion 3. Sensitive components in AIT should be well protected.* Sensitive private APIs that implement app installation should be guarded with proper access control (eg., permission for broadcast receiver). As shown in Section III, once those APIs are exposed, attackers will be able to silently install apps onto users device. Furthermore, components (eg., database files, content providers) that store installation related data should be protected. Also, database files should be kept under the installer app's private directory (internal storage).

- *Suggestion 4. User interfaces should provide more information.* Current Android design does not provide the Intent origin information, which makes the *redirect intent attack* possible. Providing more information of the app intended to be shown (eg., image of the icon, developers name/email, package name, etc.) in the appstore app before redirection, will help the users notice suspicious behaviors.

VIII. Related Work

Android vulnerabilities. PaloAltoNetworks [14] reported that an attacker can wait (using logcat: works before Android 4.1) for the permission consent dialogue and replace the target_apk once it is displayed to the user. Although they show that an attack opportunity exists for the SD-Card based installation (whose risk is already known to the app stores, given the various protection put in place to use the SD-Card), we are the first to reveal the TOCTOU window that covers the entire Step 3 and 4. Moreover, we show that even current protection of enterprises (e.g., Amazon, Qihoo, etc.) in SD-Card based installation can be defeated. Thus the security threat is significant and far-reaching (Section IV-B), not to mention the consequences that the exploits could cause, such as gaining system privileges. As evidence for the lack of understanding, even the `installPackageWithVerification` API from Google, checked only the app's AndroidManifest.xml. This can easily be circumvented by the malware using the same manifest. Our study sheds light on the TOCTOU risks of the whole AIT, its impact, fundamental cause and proposes the first solution that indeed mitigates the threat.

For other related flaws, Grace et. al. [34] discusses privacy issues on Android due to Javascript-Java bindings. They consider malicious ad libraries that exploit such bindings to perform remote runtime attacks on Java APIs without the knowledge of their host apps. However, our work reports a new technique that can exploit such bindings. We show how malware can send Intents with Javascript code to installer apps that expose a WebView with such bindings to *silently* install and uninstall apps. Also, [24, 30, 35] touch specific issues of installation: for example, how PMS checks developer signatures and assigns the UID to newly installed apps [24]. However, no effort has been made to systematically investigate installation as a transaction as did in our research.

UI Phishing attacks. Prior studies reveal various Phishing attacks through UIs [28], e.g., using a Phishing activity to cover the view of a foreground app to hijack its task flow [37]. However, none of them can directly cause a malware to be installed through an app store, because the attack app often does not have the installation permission. In addition, while prior attacks require a fake activity, ours do not. Our attack instead, changes the UI of a legitimate installer without being perceived by the user, which can lead to the installation of a malware, apparently as the result of the redirection from a legitimate app. This has never been done before. Due to such difference, previous prior [22], cannot detect nor prevent our attack. Most importantly, such a stealthy transfer of a legitimate app's own UI opens the possibility for other exploits, when the victim app can be manipulated to confuse the user.

Android side-channels. Prior studies [42, 27] infer packet data, UI states (e.g., `shared_pm`) from `/proc`. The redirect Intent attack utilizes `oom_adj`, also in `/proc`, for the first in an attack. Moreover, our purpose is just to detect UI redirections from apps to appstore apps, which is lightweight through `oom_adj`, avoiding the learning step in the prior work [27]. [41] proposed a user-level app that detects background monitoring services. However, it only protects selected apps due to high overhead. The trouble here is that many apps redirect users to appstores and all of them need to be protected.

Mitigation strategies. Many studies aim to identify malicious or suspicious apps utilizing permissions as a detection feature [31, 32, 43]. Such approach is not applicable in our case, since the adversary we consider uses a popular permission (`WRITE_EXTERNAL_STORAGE`). Even with the new runtime permission check introduced on Android 6.0, the adversary can silently gain the permission (Section III-B) by requesting once either the `READ_EXTERNAL_STORAGE` or the `WRITE_EXTERNAL_STORAGE` for a legitimate use which is likely to be granted by the user [13]. Others employ static or dynamic analysis to detect malware [23, 36, 40, 39], given known malicious behaviors. GIA is a new attack, based on a common permission and detecting it can be nontrivial. More importantly, our protection mechanisms are meant to be the last line of defense against this type of malware, even after it manages to bypass the appstore's vetting process. Although Mandatory Access Control [38, 26, 29] can help, our approach is exceedingly lightweight and effective making minimal change to the framework (Section VI-B)

IX. Conclusion

We report a study on Android app installation transaction, which led to the discovery of significant risks in this security-critical procedure. Our findings show that most installers today are not securely designed and can be exploited at every step of AIT. Particularly, the TOCTOU problem in installing apps from external storage, utilized by most appstores and system apps, essentially enables an unprivileged adversary to become a *Ghost Installer*, with the power to silently install any apps and escalate its privilege. Our research reveals the significant impacts of the GIA threats, affecting hundreds of millions of users. New techniques are developed to protect AIT against the threats. Most importantly, the study highlights the lessons learnt from our findings: *security-critical functionalities should be handled by the OS and leaving them in the hands of app developers is by no means a wise solution.*

X. Acknowledgement

We thank our reviewers for their valuable comments. This work was supported in part by National Science Foundation under grants 1223477, 1223495, 1527141, 1618493. Kai Chen was supported in part by NSFC U1536106, 61100226, Youth Innovation Promotion Association CAS, and strategic priority research program of CAS (XDA06010701). Yeonjoon Lee thanks Samsung Research America for supporting this project during his internship.

References

[1] Alexa-Ranking-APKPure. http://www.alexa.com/siteinfo/apkpure.com.

[2] Amazon Appstore: Gabriel-Knight-Sins-Fathers-Anniversary. https://www.amazon.com/Gabriel-Knight-Sins-Fathers-Anniversary/dp/B013T3AY64.

[3] Android developer website. http://developer.android.com/about/dashboards/.

[4] Android marketshare. http://www.idc.com/promo/smartphone-market-share/os.

[5] Android Storage Options. http://developer.android.com/intl/es/guide/topics/data/data-storage.html#AccessingExtFiles.

[6] Apktool. http://ibotpeaches.github.io/Apktool/.

[7] Gartner-What's next for smartphones. http://www.gartner.com/smarterwithgartner/whats-next-for-smartphones/.

[8] Ghost Installer Attacks. https://sites.google.com/site/giaprojectdemo/.

[9] Github: OSMonitor. https://github.com/eolwral/OSMonitor.

[10] GSam Battery Monitor App. https://play.google.com/store/apps/details?id=com.gsamlabs.bbm.

[11] Huawei. http://www.huawei.com/cn/.

[12] Impact of DTIgnite. http://www.digitalturbine.com/products/ignite/.

[13] Official Android Developers Documentation. http://developer.android.com/guide/topics/security/permissions.html.

[14] Palo Alto Networks. http://researchcenter.paloaltonetworks.com/.

[15] Quora-post. https://www.quora.com/I-still-have-a-lot-of-space-left-in-my-phone-memory-and-on-my-external-card-but-I-cant-download-apps-from-the-Play-Store-It-just-shows-an-error-about-a-lack-of-space-Why-is-this-happening/answer/Riccardo-Vincenzo-Vincelli?srid=kExW.

[16] samsung-updates.com. http://samsung-updates.com/.

[17] Soot. https://sable.github.io/soot/.

[18] statista.com. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.

[19] Strategy Analytics Press Releases. https://www.strategyanalytics.com/strategy-analytics/news/strategy-analytics-press-releases/strategy-analytics-press-release/2016/08/01/strategy-analytics-samsung-galaxy-s7-edge-was-world's-top-selling-android-smartphone-in-h1-2016.

[20] Xiaomi. http://www.mi.com/.

[21] Y. Aafer, N. Zhang, Z. Zhang, X. Zhang, K. Chen, X. Wang, X. Zhou, W. Du, and M. Grace. Hare hunting in the wild android: A study on the threat of hanging attribute references. In *22nd ACM SIGSAC Conference on Computer & Communications Security*, 2015.

[22] B. Antonio, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna. What the app is that? deception and countermeasures in the android user interface. In *2015 IEEE Symposium on Security & Privacy*, 2015.

[23] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *35th ACM SIGPLAN Conference on Programming Language Design & Implementation*, 2014.

[24] D. Barrera, J. Clark, D. McCarney, and P. C. van Oorschot. Understanding and improving app installation security mechanisms through empirical analysis of android. In *2nd ACM Workshop on Security & Privacy in Smartphones & Mobile Devices*, 2012.

[25] A. Bashan and O. Bobrov. Certifigate: Front door access to pwning millions of androids. BlackHat, Las Vegas, NV, 2015.

[26] S. Bugiel, S. Heuser, and A.-R. Sadeghi. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *22nd USENIX Security Symposium*, 2013.

[27] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into your app without actually seeing it: Ui state inference and novel android attacks. In *23rd USENIX Security Symposium*, 2014.

[28] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *9th International Conference on Mobile Systems, Applications, and Services*, 2011.

[29] S. Demetriou, X. Zhou, M. Naveed, Y. Lee, K. Yuan, X. Wang, and C. A. Gunter. What's in your dongle and bank account? mandatory and discretionary protection of android external resources. In *22nd NDSS*, 2015.

[30] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *16th ACM Conference on Computer & Communications Security*, 2009.

[31] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *ACM Conference on Computer & Communications Security*, 2011.

[32] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *20th USENIX Security Symposium*, 2011.

[33] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: Validating ssl certificates in non-browser software. In *2012 ACM Conference on Computer & Communications Security*, 2012.

[34] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *5th ACM Conference on Security & Privacy in Wireless & Mobile Networks*, 2012.

[35] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In *18th ACM Conference on Computer & Communications Security*, 2011.

[36] C. Mann and A. Starostin. A framework for static detection of privacy leaks in android applications. In *27th Annual ACM Symposium on Applied Computing*, 2012.

[37] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu. Towards discovering and understanding task hijacking in android. In *24th USENIX Security Symposium*, 2015.

[38] S. Smalley and R. Craig. Security enhanced (SE) android: Bringing flexible MAC to android. In *20th NDSS*, 2013.

[39] L. Vigneri, J. Chandrashekar, I. Pefkianakis, and O. Heen. Taming the android appstore: Lightweight characterization of android applications. *CoRR*, 2015.

[40] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. Appintent: analyzing sensitive data transmission in android for privacy leakage detection. In *2013 ACM SIGSAC conference on Computer & communications security*, 2013.

[41] N. Zhang, K. Yuan, M. Naveed, X. Zhou, and X. Wang. Leave me alone: App-level protection against runtime information gathering on android. In *2015 IEEE Symposium on Security & Privacy*, 2015.

[42] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt. Identity, location, disease and more: Inferring your secrets from android public resources. In *2013 ACM SIGSAC Conference on Computer & Communications Security*, 2013.

[43] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *19th NDSS*, 2012.