Upgrading Your Android, Elevating My Malware: Privilege Escalation Through Mobile OS Updating

Luyi Xing*, Xiaorui Pan*, Rui Wang†, Kan Yuan* and XiaoFeng Wang*
*Indiana University Bloomington
Email: {luyixing, xiaopan, kanyuan, xw7}@indiana.edu
†Microsoft Research
Email: ruiwan@microsoft.com

***Abstract*—Android is a fast evolving system, with new updates coming out one after another. These updates often completely overhaul a running system, replacing and adding tens of thousands of files across Android's complex architecture, in the presence of critical user data and applications (*apps* for short). To avoid accidental damages to such data and existing apps, the upgrade process involves complicated program logic, whose security implications, however, are less known. In this paper, we report the first systematic study on the Android updating mechanism, focusing on its Package Management Service (PMS). Our research brought to light a new type of security-critical vulnerabilities, called *Pileup* flaws, through which a malicious app can strategically declare a set of privileges and attributes on a low-version operating system (OS) and wait until it is upgraded to escalate its privileges on the new system. Specifically, we found that by exploiting the Pileup vulnerabilities, the app can not only acquire a set of newly added system and signature permissions but also determine their settings (e.g., protection levels), and it can further substitute for new system apps, contaminate their data (e.g., cache, cookies of Android default browser) to steal sensitive user information or change security configurations, and prevent installation of critical system services. We systematically analyzed the source code of PMS using a program verification tool and confirmed the presence of those security flaws on all Android official versions and over 3,000 customized versions. Our research also identified hundreds of exploit opportunities the adversary can leverage over thousands of devices across different device manufacturers, carriers and countries. To mitigate this threat without endangering user data and apps during an upgrade, we also developed a new detection service, called SecUP, which deploys a scanner on the user's device to capture the malicious apps designed to exploit Pileup vulnerabilities, based upon the vulnerability-related information automatically collected from newly released Android OS images.**

I. Introduction

Mobile operating systems (OSes) are evolving quickly. Every a few months, major updates or new overhauls of entire systems are made available, bringing to mobile users brand new apps and enriched functionalities. Conventional wisdom is that such a vibrant ecosystem benefits the phone users, making mobile systems more usable and also more secure, through timely plugging loopholes whenever they are found. Indeed, for years, major smartphone vendors and system/software developers leverage convenient updating mechanisms on phones to push out fixes and enhance existing protection. However, with such updates becoming increasingly frequent (e.g., every 3.4 months for all 19 Android major updates [5]) and complicated (e.g., hundreds of apps being added or replaced each time by hundreds of different Android device vendors), questions arise about their security implications, which have never been studied before.

New challenges in mobile updating. Security hazards that come with software updates have been investigated on desktop OSes [45], [37]. Prior research focuses on either compromises of patches before they are installed on a target system [26] or reverse-engineering of their code to identify vulnerabilities for attacking unpatched systems [40]. The reliability of patch installation process has never been called into question. For a mobile system, this update process tends to be more complex, due to its unique security model that confines individual apps within their sandboxes and the presence of a large amount sensitive user data (e.g., contacts, social relations, financial information, etc.) within those apps' sandboxes. Every a few months, an update is released, which causes replacement and addition of tens of thousands of files on a live system. Each of the new apps being installed needs to be carefully configured to set its attributes within its own sandboxes and its privileges in the system, without accidentally damaging existing apps and the user data they keep. This complicates the program logic for installing such mobile updates, making it susceptible to security-critical flaws. Also adding to this hazard is fragmentation of mobile OSes, particularly Android, the most popular system. Multiple official Android versions (from Froyo to Jellybean) co-exist in the market [3], together with thousands more customized by different vendors (Samsung, LG, HTC, etc.). Those versions are slowly but continuously updated to higher ones [3], leaving the potential adversary a big window to exploit their update installation process, should its security flaws be uncovered. With the importance of this issue, little has been done so far to understand it, not to mention any effort to mitigate the threat it may pose.

Menace of Pileup. In our research, we conducted the first security analysis of mobile updating, focusing on Android Package Manager as a first step. Our study brings to light a new category of unexpected and security-critical vulnerabilities within Android's update installation logic. Such vulnerabilities, which we call *Pileup* (privilege escalation through updating), enable an unprivileged malicious app to acquire system capabilities once the OS is upgraded, without being noticed by the phone user. *A distinctive and interesting feature of such an attack is that it is not aimed at a vulnerability in the current system. Instead, it exploits the flaws in the updating mechanism of the "future" OS, which the current system will be upgraded to*. More specifically, through the app running on a lower version Android, the adversary can strategically claim a set of carefully selected privileges or attributes only available on the higher OS version. For example, the app can define a new system permission such as `permission.ADD_VOICEMAIL` on Android 2.3.6, which is to be added on 4.0.4. It can also use the shared *user ID* (UID) [17] (a string specified within an app's manifest file) of a new system app on 4.0.4, its package name and other attributes. Since these privileges and attributes do not exist in the old system (2.3.6 in the example), the malicious app can silently acquire them (self-defined permission, shared UID and package name, etc.). When the system is being updated to the new one, the Pileup flaws within the new Package Manager will be automatically exploited. Consequently, such an app can stealthily obtain related system privileges, resources or capabilities. In the above example, once the phone is upgraded to 4.0.4, the app immediately gets `permission.ADD_VOICEMAIL` without the user's consent and even becomes its owner, capable of setting its protection level and description. Also, the preempted shared UID enables the malicious app to substitute for system apps such as Google Calendar, and the package name trick was found to work on the Android browser, allowing the malicious app to contaminate its cookies, cache, security configurations and bookmarks, etc.

With the help of a program analyser, our research discovered 6 such Pileup flaws within Android Package Manager Service and further confirmed their presence in all AOSP (Android Open Source Project) [1] versions and all 3,522 source code versions customized by Samsung, LG and HTC across the world that we inspected. The consequences of the attacks are dire, depending on the *exploit opportunities* on different Android devices, that is, the natures of the new resources on the target version of an update. As examples, on various versions of Android, an upgrade allows the unprivileged malware to get the permissions for accessing voicemails, user credentials, call logs, notifications of other apps, sending SMS, starting any activity regardless of permission protection or export state, etc.; the malware can also gain complete control of new signature and system permissions, lowering their protection levels to "normal" and arbitrarily changing their descriptions that the user needs to read when deciding on whether to grant them to an app; it can even replace the official Google Calendar app with a malicious one to get the phone user's events, drop Javascript code in the data directory to be used by the new Android browser so as to steal the user's sensitive data, or prevent her from installing critical system apps such as Google Play Services [8]. We performed a measurement study on those exploit opportunities, which shows how they are distributed across Android versions, countries, carriers and vendors. Particularly, we found that customized OSes are highly susceptible to the Pileup attacks, due to a large number of system capabilities they bring in for each upgrade. We have reported our findings to major Android-device vendors, e.g. Google, and are helping them fix the issues. Demos of our exploits are online [14].

Secure updating. The Pileup vulnerabilities are critical, highly pervasive and also fundamental. It is caused by a conservative strategy device manufacturers have to take to avoid replacing resources of unknown origins on the existing system (Section II-A) during an upgrade, for the purpose of protecting their users' data and installed high-version apps. As discussed before, we found the presence of the problem in thousands of Android images we scanned and highly suspect that all Android devices are vulnerable to our attacks. Given that near a billion Android devices [11] are out there, simply patching all of them within a short period of time is unrealistic. Also, given the fundamentality of the issue, any less than well-thought-out fixes will easily lead to serious side effects, including damages to user data or installed high-version apps. To better understand the problem and practically mitigate its threat, we developed a Pileup detection service, called *SecUP*. SecUP provides a scanner app to inspect installed Android application packages (APKs) on an Android device, in an attempt to identify those that will cause privilege escalations during an update. It includes a mostly automated vulnerability detector built upon *VeriFast* [18], a program verification tool for Java, that discovers the Pileup flaws within the source code of different Android versions, and a threat analyser that automatically scans thousands of OS images to find out all the exploit opportunities related to these flaws. We built a database that documents all the opportunities generated as the result of the analysis, which is used by the scanner to check installed APKs. We utilized SecUP to perform the aforementioned measurement study and also evaluated its effectiveness. Our study shows that the approach can catch Pileup risks and avoid incriminating the new version of a legitimate app installed on an Android device before OS upgrading.

Contributions. We summarize the contributions of the paper as follow:

- *New findings*. We performed the first systematic study on the security risks in mobile updating mechanisms through analyzing Android Package Management Service, and discovered Pileup, a new type of privilege escalation vulnerabilities. Successful exploits of the vulnerabilities can have devastating consequences. Our research confirmed the presence of those flaws in all official Android versions and all 3,522 customized systems from major smartphone vendors, indicating that *all* Android versions could be affected by this problem. We further conducted a measurement study over 3,549 factory images from Google and Samsung, and discovered tens of thousands of attack opportunities across different Android versions, countries, carries and vendors, each of which enables a knowledgable adversary to acquire system capabilities automatically during an upgrade.

- *New techniques*. We developed a new service for automatically identifying the Pileup risks on a mobile OS. Our approach can continuously scan new Android versions emerged, find out new vulnerabilities or exploit opportunities they introduce, and conveniently detect related malicious apps on Android devices without undermining its utility.

Roadmap. The rest of the paper is organized as follows: Section II provides the background information about mobile updates and discusses its potential security risks; Section III elaborates the Pileup vulnerabilities we discovered; Section IV presents the design and implementation of the SecUP service for detecting Pileups; Section V reports our evaluation of SecUP and a measurement study on the consequences of Pileups using our service; Section VI discusses the limitation of our research and potential future directions; Section VII compares our work with related prior research and Section VIII concludes the paper.

II. Hazards in Mobile OS Updating

A. The Problem

Mobile OSes are characterized by their quick paces of updating, which enable smartphone vendors to continuously push their new services to their customers and timely fix all the problems that show up. Different from desktop OSes, whose whole system upgrades happen less frequently, mainstream mobile systems like Android and iOS rapidly elevate their customers' OSes from one version to another, patching and adding tens of thousands of system files each time (e.g., 15,525 files when upgrading Android 4.0.4 to 4.1.2). This updating process is also carefully designed to inherit all user information on the old system and operate in a way completely transparent to the user: typically, all one needs to do here is just to push a button, and the new OS will then be downloaded from the Internet and installed on her phone without disrupting the operations of her favorite apps and messing up her settings and data. Such a design, however, significantly complicates the update mechanisms on mobile devices. Understanding the security implications of this new complexity is the purpose of our study, with our focus on Android as the first step.

Since September 2008, 19 Android official versions (from 1.0 to 4.4) have been released. New updates are available almost every three months [5]. Figure 1 illustrates the sequence of such updates. In addition to these AOSP versions, phone manufacturers also provide their customized OSes, oftentimes, for different carriers (Verizon, AT&T, T-Mobile, etc.) and different countries. For example, Samsung has so far released over 10,000 different versions to serve its customers world wide. Many of them co-exist on today's Android market, causing what is so called *Android fragmentation* [3]. On the other hand, these versions are being steadily upgraded to newer ones: it is reported that within the recent 6 months, the market share of Jelly Bean (Android 4.1+) has increased from 25% to 48.6% [2]. This trend is supported by the increasing convenience in updating on more recent devices. In our research, we analyzed popular Android devices released by major manufacturers in the past three years, including Google, Samsung, LG, HTC, Motorola and Sony Erisson, and found that all of them can be easily upgraded through WiFi or 3G, with a single click on the "update" button. From security viewpoints, however, this upgrading process could leave a big window to the adversary who wants to exploit vulnerabilities within Android updating mechanisms. How serious the threat is depends on what can be found from the mechanisms. To better understand this issue, we need to take a look at how the updating works, as described below.

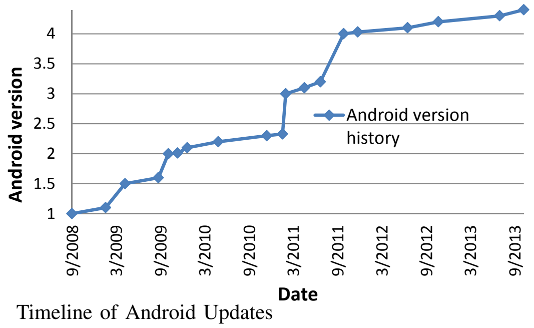


Fig. 1. Timeline of Android Updates

Android updating. The technical details for the Android updating process have not been officially released. All we can learn about it is from third-party reports and our manual analysis based on system logs, runtime observations and related source code. Here is the picture we come up with.

After the user clicks on the system update button under Android Setting, an upgrading image (called "Over the Air" or OTA) is downloaded from the device manufacturer's servers[1]. The image includes `bootloader.img`, other system files and a set of Android application package files distributed across different directories. Some of them are patches to the existing files (on the old system), which typically end with "`.p`" (e.g., `radio.img.p`). Once receiving the image, the Android device reboots into a *recovery* mode, where it verifies the authenticity and integrity of the image and then replaces existing system files such as bootloader, Package Manager Service and APKs under the `system` directory with the new ones. After this is done, the device reboots into the new OS and continues to update other system components through different Android services, including Activity Manager, Service Manager, User Manager Service, Package Manager Service, Input Manager, etc. Of particular interest here is Package Manager Service (PMS), which we investigated in our research.

PMS is an Android service for installing, upgrading, configuring and removing application packages. When it comes to upgrading, the service installs or reinstalls all system apps (new and existing ones from the old OS) under `/system/` directory, and then existing third-party apps (user installed) under `/data/app` onto the new OS. When installing an app, PMS registers its permissions[2], shared UID, activities, intent filters, actions, services and others. What complicates this installation process is the presence of duplicated attributes (e.g., package names, shared UIDs, etc.) and properties (e.g., permissions). In this case, PMS needs to decide which one to install. For this purpose, it builds a data structure `mSettings` to record all such information about existing apps on the old OS, which includes a group of lists (in the `HashMap` type) such as `mPackages`, `mUserIds`, `mSharedUsers`, `mPermissions`, etc. Whenever a new system package is about to be installed or its properties to be registered, PMS first looks up these lists to find out whether another package with the same attributes like package names[3] or duplicated properties like permissions already exists in the old system. This happens, for example, when a high-version system app is installed on the old OS to replace its low-version counterpart. Once a new package or its property is found to be already in the old system, the decision on which one to keep is made case by case, based upon the nature of the package or the property. For example, an app from the updating image will replace an existing one if the former has a higher version number than that of the latter.

What can go wrong. This installation process has been

[1] Some old devices are required to be physically connected to a PC through their USB to get access to the updating package.

[2] Note that on Android, all the permissions (i.e., privileges for accessing system resources) are declared by certain apps. Particularly, all AOSP permissions are defined by the package `android` and other system apps.

[3] The user can install a high-version system app like Google Plus on a low version Android. In this case, the app is placed under `/data/app` directory, instead of `system` where all system APK files are placed.

designed to preserve user data and avoid improperly replacing existing properties, which is of paramount importance to an update that happens to a live system. However, achieving the goal is by no means easy. Compared with Linux, Android is much more complex, with its layered architecture designed to shield the user from low-level details. Particularly, on Android, each app is confined within its sandbox, together with user data it guards. Its attributes and properties (i.e., permissions, package names, shared UIDs, etc.) are system-wide, uniquely identifying its package and its privileges within the system. This complicates an update process in the presence of duplicated package attributes or properties: not only does PMS need to check different conditions on the packages (with such attributes/properties) in conflict before choosing the right one (either the original one on the old OS or the new one within the update image) to install, but it may also has to carefully combine two apps together, for example, keeping the data from the old version while using the code of the new one. The program logic here can become very complicated, and thus error-prone, as discovered in our research (Section IV). Also problematic is that for the sake of simplicity, PMS uses the same program logic for both system upgrading and normal app installation: that is, the decision on whether to install a new system app or register its properties (e.g., permissions) is always made in the same way as when a third-party app to be installed is found in conflict with an existing app.

Once PMS keeps a wrong attribute or property (one introduced by a third-party app, instead of its system counterpart), the consequence can become very serious. Android often refers to privileges, resources and data by their names. As a result, the third-party package attribute or property, which bears the name of its system counterpart, can be elevated to a system one during the updating shuffle-up where all apps are installed or reinstalled, and all system configurations are reset. Also, when two apps from old and new systems are merged as described above, security risks can also be brought in when the one on the original system turns out to be malicious. To avoid these and other pitfalls in the updating process, its design needs to be carefully thought out. This cannot be done without an in-depth understanding about what can go wrong here, a question our research attempts to answer.

B. Adversary Model

We consider an adversary with his malicious apps installed on the victim's Android devices. In Section III-E, we show that such malware can be uploaded to Google Play and third-party Android markets, or disseminated to Android users through other channels such as email attachments. Also, the apps here can appear less dangerous than some legitimate apps on their target Android versions, simply because they may not ask for dangerous permissions (which they wait for the next update to get). Of course, the threat becomes particularly serious when there are a lot of old mobile OSes that are slowly but steadily updated to new ones, and such updates come with addition of many security-critical privileges and capabilities. This is exactly what is happening in the Android ecosystem.

III. Pileup Exploits

In this section, we elaborate the Pileup vulnerabilities discovered from Android PMS and our exploits on them. Some of these flaws were first found manually, which motivated this research, and the others were first caught by our Pileup detector (Section IV). All the problems we found are essentially a type of broadly defined *privilege escalation* weaknesses, by which we refer to not only the situation where an unauthorized party gains elevated *access* to protected resources but also when it acquires elevated capabilities to *deny* authorized parties' access to the resources. We present those problems below.

A. Permission Harvesting and Preempting

At the center of the Android security architecture is its permission and sandbox model. Each app runs within its own sandbox, separated from other apps. To gain any additional capabilities that would impact other apps or the OS, the app needs to explicitly request a permission (within its `manifest.xml`) before it is installed. These permissions are used to guard resources such as Internet, camera, storage, etc. They are categorized into *protection levels* [4], among which most common are *normal* (automatically granted to apps when requested), *dangerous* (granted based upon the user's consent), *signature* (granted to apps signed with the same certificate as the one that declares the permission), *System* (granted to system apps on the system image) and a combination *signatureOrSystem* (granted to system apps or those signed with the same certificate). All the standard (AOSP) permissions are declared by system packages *android* and others located under */system/*. Third-party apps can also define their own permissions, which can be requested by others.

In our research, we found that the program logic within PMS for handling the permissions inherited from the old system is problematic, which allows a malicious app to automatically gain dangerous or even system and signature permissions added by the new OS without the user's awareness, or even become the source package of these permissions with the power to set their protection levels and descriptions. Here we elaborate how this can happen.

Existing/conflicting permissions. As discussed above, right before an app is about to be installed, PMS needs to check the permission it requests, and most of the time, turns to the device user for consent. All such requests are specified in the app's manifest file. What is interesting here is that when PMS goes through the manifest, it ignores all those it fails to recognize, and never reports them to the user. This design, presumably, is for handling the third-party apps that are not well implemented. The problem is that the adversary can take advantage of this opportunity, letting his malicious app ask for a set of dangerous permissions only declared on a higher Android version without being noticed by the device user.

When the OS is being upgraded to the higher version, PMS first installs all new and existing system apps and registers the permissions they declare, and then moves on to install third-party apps from the old OS. When it is the malicious app's turn, this time, PMS recognizes all the permissions it requests, including the ones that come with new system packages and have just been defined. In this case, everything that the app asks for is just silently granted, since these permissions are with an existing app and supposed to have already been approved by the user. However, as we know, the truth is that those new dangerous permissions have never been identified by the system before the updating and are thus never known to the user. Exploiting this weakness, a malicious app can "harvest" permissions (i.e., requesting them on the old system) and wait until an update to get them. Note that this attack can only get the app dangerous level permissions, since signature and system permissions declared by system apps cannot be granted to the third-party app and an attempt to get them will be identified during permission registration.

However, there is a way to get the permissions at signature and system level: the malicious app can preempt the permissions from the new system and simply define them when it is installed. Again, since there is no such permissions on the old system, the OS just lets the app declare them, which includes specification of the permissions' protection levels and descriptions. This process does not need the user's intervention at all, as all these permissions are used to protect the sources that come with the app. During an update, PMS reads all existing permissions (on the old system) into the list `mSettings.mPermissions`. When it registers a new permission defined by a new system package, it first goes through the list: once this permission has been found on the list and the package that first defines it (on the old system) is different from the current one, the permission definition part (for the system package) is skipped. As a result, the old permission, which has been declared by the malicious app, is automatically elevated to the one that guards new system resources. That is, whenever any app needs to access the resources, a related Android service will check whether the app has this permission. Note that not only does the malicious app that defines the permission get the permission (even at the signature and system level), it is also the party that specifies the protection-level of the permission and its descriptions. In other words, the adversary can lower a system permission to a normal one and arbitrarily set its descriptions. Also, on the new system, once the malicious app is uninstalled, this whole permission is removed from the OS. There is no way for any other app to request it and therefore none gets the privilege to touch the resources under its protection.

Attacks and consequences. Our research shows that the permission harvesting and preempting vulnerabilities exist in *all* official Android versions and *all* 3,522 customized source code versions by Samsung, LG and HTC that we inspected (Section V). What the adversary can get from exploiting these flaws depends on the permissions added by the target versions of different updates. In our research, we implemented an app to play those tricks when our Google Nexus S and Galaxy Nexus phones were upgraded from 2.3.6 to 4.0.4, then to 4.1.2, 4.2.2 and 4.3 consecutively. Our app successfully obtained and also preempted all the permissions we tested, with some of them elaborated in Table I. In the table, we also present the permissions in lower Android versions that are also vulnerable to our attack. For all the system, signature and dangerous permissions we obtained, our app deliberately lowered their protection levels to normal, so other parties can get them automatically without user's explicit consent. An example is `certinstaller.INSTALL_AS_USER`, a signature permission for installing the root certificate under the `download` directory on an SD card. We further utilized the permissions `READ_PROFILE` and `READ_CALL_LOG`, both at the dangerous level, to get unauthorized access to the profile and the call log on Android 4.0 and 4.1 respectively.

Permission Name	**Protection Level**	**Android Version**	**Description (allows to)**
Mount_format_ FileSystems	signature OrSystem	1.5	format removable storage
Use_credentials	dangerous	2.0	request authentication tokens
GoogleVoice.SMS	dangerous	4.0	receive Google Voice messages
Retrieve_ window_content	signature OrSystem	4.0	retrieve content of entire active window except passwords
Send_sms_ no_Confirmation	signature OrSystem	4.0	send SMS messages without confirmation
Start_any_activity	signature	4.1	start any activity, ignore permission or exported state
Grant_permissions	signature	4.1	grant specific permissions for apps
Plus.Picasa	dangerous	4.2	access photos in Google Photos
Across_users	signature OrSystem	4.2	violate protection between users
Access_notification	signature OrSystem	4.3	retrieve and clear notifications including those of other apps

TABLE I. SELECTED PREEMPTED PERMISSIONS

As another example, we preempted the permission for Google Cloud Messaging (GCM) [7] to intercept its *Push* messages. GCM is a service that helps developers send data from their servers to their Android apps. The messages delivered through this service can contain up to 4KB of payload data. System apps employing GCM to push messages include Google Plus, Google Hangout, Gmail, Google Voice, etc. To receive the *Push* messages, an app must register a signature permission `packageName.permission.C2D_MESSAGE`. In our attack, our app preempted the GCM permission on our Google Nexus phones and got the permission to eavesdrop on the sensitive *Push* messages delivered to affected system apps, which had a significant security implication.

B. Shared UID Grabbing

Android security has been built upon Linux user protection. Each app running on Android is assigned a UID, which prevents it from accessing other apps' information assets. An exception, however, is provided for those bearing the same *shared UID*, an attribute that comes with individual app. Specifically, two apps can declare in their manifests an identical shared UID under `android:sharedUserID` [17], a constant string, which causes the OS to assign them the same UID (a number) when they are installed, if they are also signed by the same party. When this happens, these apps can access each other's data and even execute in the same process.

In our research, we found that a malicious app on a low-version Android can declare the shared UID used by a system app on a high-version system. During an update, this forces PMS to skip the installation of the new app, which gives the adversary an opportunity to replace the app with a malicious one. Following we elaborate this Pileup problem and its consequences.

Shared UID handling. As discussed before, during an update, PMS is invoked to go through all APK files under `system` directory and then `/data/app/` to install new apps and reinstall existing apps one by one. This procedure excludes the third-party app carrying a system package's name, simply because after the system app has been installed, the third party app can no longer be reinstalled, due to the conflict of package names. However, we show here that PMS will handle this situation differently when the third-party app also has a shared UID. When installing a system app, PMS creates a class instance `pkgSetting` that holds the app's setting information. The content of the data structure normally comes from the app itself. However, when PMS looks up the list `mSettings` and finds a conflict in package name (an existing app with an identical package name from the old OS), it will look at both packages' shared UID settings: if they use the same shared UID (including empty ones), the structure `pkgSetting` will be loaded from the existing app. Then in the case of non-empty share UIDs, PMS verifies their app signatures to check whether they are signed by the same party. When this is not true, the new system app will not be installed so the existing one could be installed later. Presumably, this treatment is meant to be conservative, as Google and other vendors have no idea whether the existing app and its data is useful to the user. Note that on the same device, by no means two apps signed by different parties should share their UIDs, which will completely open them to each other in terms of their individual information assets. However, this conservative treatment can be exploited by the adversary, who can craft an app bearing the same package name and shared UID as the system app from a higher-version OS. During upgrading, the app can block the installation of the system app. This can have serious consequences, which we will discuss later.

This problem was first discovered through a manual check of the code, which had motivated this research. Later, when we ran our Pileup detection tool (Section IV) on PMS, it turned out that the package name is unnecessary for the attack. Interestingly, even in the case that PMS does not identify any package conflict and thus `pkgSetting` contains the settings from the system package itself, as long as the new package configures a shared UID, Android will retrieve from `mSettings` all existing packages with the same Shared UID and inspect their signatures to find out any inconsistency with that on the new package. If one such app is signed by a different party, the new package will not be installed. In this way, a malicious app only needs to preempt the new package's shared UID to knock it out of the system.

When the updating process moves Android into the recovery mode, all the APKs files under the `system` directory are replaced with the new ones[4] (Section II). As a result, once a new system app fails to install, it becomes completely missing on the new OS, as its old version under `system` on the original system (before the upgrading) has already been overwritten. This gives the adversary an opportunity to install a malicious one in its replacement, which can be done by downloading another package through the existing malicious one, making it look like part of the upgrading.

Attacks and consequences. In our research, we installed an app on a Nexus S with Android 2.3.6. The app claimed the shared UID `com.google.android.calendar` on 4.0.4. After the upgrading, it successfully blocked the installation of the new official calendar app without user's awareness. Given the original calendar app was already removed, we filled this blank with a malicious calendar downloaded through the attack app, which set a filter for receiving all the intents for adding events to the calendar. This malicious calendar can be made to have the same user interface as the official calendar app. Note that should the official calendar app still be there, the user would be notified that two apps were monitoring the same intents and asked to choose one of them, while in our case, our app was the only one expecting the events and therefore such a notification did not come out. We have a video demo posted online [14] to show how our app stealthily gets private user events and schedules. The same exploit can also work on other Android system apps, as illustrated in Table II.

sharedUID	Package Name	Device Model	Carrier/ Country	Android Version
uid.nfc	com.sec.surfsetprop	SGH-T869	TMB/US	4.0.4
uid.platform	com.sec.widget	GT-P7300	SER/Russia	4.0.4
uid.graphics	com.samsung.reader	GT-P6800	SER/Russia	4.0.4
vmware.mvp	vmware.mvp	SCH-I535	VZW/US	4.1.2
uid.widget	sec.app.launcher	GT-I8160	LUX/Luxemburg	4.1.2
com.c	chinaunicom.cloud	EK-GC100	CHU/China	4.1.2
scloud.sync	sCloud.datasync sCloudSyncBrowser sCloudSyncContacts	GT-I9300	CHU/China	4.1.2

TABLE II. EXPLOIT OPPORTUNITIES OF SHARED UID GRABBING ON SAMSUNG DEVICES

C. Data Contamination

All the precautions made by PMS during an update are for the purpose of avoiding any potential damage to the device user's existing assets, particularly her data. Indeed, even when PMS decides to replace an existing app with a new one, it still carefully merges the former's data into the latter during its installation, making sure that it does not mess up the user's configurations and personal information. Such a treatment, however, provides the adversary another opportunity to escalate his privilege on the infected system, this time through the data the malicious app left to the new system app.

Reuse of existing data. Android keeps the data for both system and third-party apps under directory `/data/data/PackageName`, which is owned by a unique Linux UID in the absence of UID sharing. An app is only allowed to access the information under its own data directory. During an update, PMS will compare the UID recorded within `pkgSetting` with that of the existing data directory associated with the package name of the new system app being installed, keeping the directory when they match and clear the directory otherwise. The UID within `pkgSetting` is typically a new one assigned by Linux to the new system app. However, when there is another app installed on the old OS with the same package name and Shared UID, as we explained before (Section III-B), `pkgSetting` will contain the existing app's information. The UID within `pkgSetting` is naturally the owner of its data directory under the inspection. When this happens, if the shared UIDs of the two apps are both empty, the new system app is installed but the data of the existing app is left and incorporated into the new app. Therefore, a malicious app that bears the package name of a new system app when neither has a shared UID, will be able to contaminate the latter's data through OS upgrading. (Note that the malicious app will not be reinstalled successfully later for the package name conflict which means it will not exist in the new OS). The consequences here are serious and diverse, depending on

[4] Note that in the case that an Android user installs a high-version system app on a low-version OS, that package is placed under the `/data/app` directory and therefore will not be replaced in the recovery mode.

the natures of the new apps. Here we describe an end-to-end attack on the Android default browser.

Attacks and consequences. The package name of the official browser for Android 2.3 is `com.android.browser`. After the system is upgraded to Android 4.0, the package name is updated to `com.google.android.browser`. In our research, we exploited this discrepancy in its package names[5] to implant an attack app on 2.3.6 that used the new browser package name. An update to 4.0.4, therefore, caused our app to be removed and the old browser to cease to exist but rendered the new browser using our app's data directory. Under the directory are the browser's databases (`.db` files) that keep cookies, security configurations and bookmarks, etc. and also cache for web pages, scripts and others. The database files from the old OS will be replaced if their versions are lower than that specified by the new browser. In our attack, however, we deliberately built high-version `.db` files into our attack app, thereby leaving all our content untouched. Within those files, we put the cookies for our own Google, Dropbox, Facebook and other accounts and successfully caused login cross-site request forgery (CSRF) attacks, in which the phone user unconsciously logged into our account, and her searching terms and other personal data were therefore made available to us. Also, we dropped web pages with malicious JavaScript in the cache. As a result, whenever the browser visits the target websites, our cached pages are loaded and the script get access to the user's cookies and other web data. Moreover, the Android browser includes in a whitelist of websites allowed to access the device's geolocation. Our app contaminated this list and added our malicious website onto it. As a result, a visit to the website automatically discloses the phone user's geolocation to us. More seriously, through our app, we even tampered with the browser's built-in bookmark list. On the list are a set of website-URL pairs, which we modified to point to the sites under our control. For example, we replaced the URLs for Google, Yahoo, Facebook, Twitter, etc. with the links to our sites. Note that this bookmark list can also include what the user sets on her 2.3.6 device. As the result of this exploit, whenever the user uses her bookmarks, she is always directed to our websites. Video demos of the attacks are here [14].

Again, such attacks are not limited to browser. Table III illustrates some of the other opportunities we found from Google Nexus and Samsung devices.

Update Version	Package Name		Update Version	Package Name
2.3 -4.0	android.exchange android.keychain google.gsf.login google.apps.plus		4.0 -4.1	com.dropbox samsung.gmail
			4.1 -4.2	sec.safetyassurance samsung.accesscontrol

TABLE III. OTHER EXPLOIT OPPORTUNITIES FOR DATA CONTAMINATION

D. Denial of Services

As discussed in Section III-A and Section III-B, the Pileup flaws within PMS lend a malicious app an elevated capability to deny a mobile system new resources added through an update. Here we describe two other vulnerabilities that can also prevent PMS from installing new system resources.

Exploiting permission tree. On Android, a permission typically can only be defined before an app has been installed. An exception is when the app specifies a *permission tree* [13], which is the base name (root) of a tree of permissions. An app can define such a base name in its manifest file so as to claim the ownership of all permission names within the tree. For example, given a base `com.example`, permissions like `com.example.math1`, `com.example.math1.add`, `com.example.math2` etc. all belong to the tree. Once declaring the tree, the app controls the whole name space defined by the root, and can then add individual permission within the tree during its runtime.

As discussed before, Android does not allow the existence of two permissions with the same name. What the adversary can do here is to define a permission tree, covering permissions to be added by a new Android version, through his app on a low-version OS. Those permissions are not defined yet but their names are all owned by the malicious app. During an update, PMS checks the `mPermissiontrees` list under `mSettings` before registering any new permission. Whenever a permission's name is found to be covered by an existing tree declared by a different package from the system package that is registering the permission, its registration process fails. Interestingly, since by default, PMS assigns every permission a `signature` protection-level before changing it to the actual levels at the end of a registration, this disruption produces a signature permission no one can get[6], and thus prevents legitimate apps from accessing the resources the permission guards. As one example, `permission.ADD_VOICEMAIL`, the permission required to add voicemail became inaccessible to legitimate apps due to our attack when updating from Android 2.3 to 4.0. Also interestingly, we found that even in the presence of a permission tree defined by a system app already on the old system, the malicious app can still prevent the new system from registering any new permissions covered by the existing, legitimate tree. The key issue here is that Android allows our app to define another permission tree that also covers the existing one. For example, even though the system app `google.android.gsf` on Google Nexus already defines the tree `google.apps.permission.GOOGLE_AUTH`, our app can still declare a permission tree `google.apps.permission`. As a result, any related new permissions the new system adds will not be able to register even if they are within the name space of `google.apps.permission.GOOGLE_AUTH`, as they are also covered by our permission tree. Also note that the device user will not be aware of this problem, as Android never notifies her the failure of the registration. We found that this vulnerability can be exploited on every single Android version, given the large number of new permissions added by each update, as studied in section IV-C.

Blocking Google Play Services. Google Play Services is a critical package introduced by Android 4.0 on Google compatible devices. It is used to update apps from Google

[5] Otherwise, we would not be able to install a malicious app, as two packages on the same OS are not allowed to have the same package name.

[6] An eligible app should also be signed by the developer who defined the permission.

Play, offering key supports for authentication, synchronized contacts, access to user private settings and location-based services [8]. Its absence will disable some important system apps and third-party apps. We found that when a device is upgraded from Android 2.3 to 4.0 on Nexus S, this critical service is actually not part of the update image as a system app. Instead, after installing all new and existing apps, Android downloads this package from the Internet and installs it as third-party app. As a result, a malicious app on 2.3.6 using the same package name will stop the installation of the service, simply because PMS cannot have two packages with the same name coexist on the OS. Actually, this also happens when the app includes a content provider with an authority name in conflict with that of the new package. PMS in this case will also skip the new package. We implemented both attacks and successfully blocked Google Play Services. In this way, any app relying on it will not work. For example, when opening Google Plus on our Nexus S, the app crashed and the OS prompted to the user a message "Google Play Services, which some of your apps rely on, is not supported by your device", which is very misleading. Also, many popular apps rely on Google Cloud Messaging [7], which is based on Google Play Services, to `push` messages. These apps include all top 10 free apps in the `social` category on Google Play. They all crash or function unexpectedly due to our attack. The root cause of this problem is that a new package, critical to other system apps and third-party apps, is not included in OS updating image as a system app, but downloaded as a third-party app.

E. Discussion

Other Pileup flaw. A unique feature of all Pileup vulnerabilities is that the consequences of any successful exploits on them hinge upon the types of new capabilities the adversary can acquire through them during an update. This sounds that as long as Android does not offer any security-critical new functionalities related to these flaws, their risk levels seem to be low. The problem is that the Pileup flaws we found have always been there since the first Android version, and also within every manufacturer-customized system (Section V). Therefore, even for those that do not bring in any harms on the existing systems, they are essentially ticking time bombs waiting for explosions (when the right functionalities are added by Google or device manufacturers one day). Here we describe one such flaw discovered in our study.

When PMS detects a package name conflict when installing a system app, in the case of both apps having identical shared UID, not only does it incorporate the existing app's data into the new system app, it also copies the whole settings of the existing app into a data structure `pkg.mExtras`. This structure partially contributes to the configuration and inspection of the new app's settings. For example, it is used to check whether a permission is given to the app. Under our attack, the content of the structure can be manipulated by the attack app that takes the package name and Shared UID of the new system app. Even though so far, not much can be done to exploit this issue for escalating the adversary's privilege on the user's device, the evolution of PMS, which has been adjusted in almost every new Android release, could one day bring in an opportunity to make the problem more serious.

Malware distribution. To understand whether malicious apps exploiting Pileup flaws can be easily disseminated in practice, we attempted to upload them to both Google Play and third-party Android marketplaces. It turns out that all the apps relying on the seizure of new permissions, UIDs and content provider names have no problem being added to Google Play. Therefore, the attack code for most exploits (Section III-A, III-B and III-D) can be conveniently distributed and delivered to Android users. On the other hand, Google does check package names and disallows those having name conflicts with existing packages to be uploaded. However, such a restriction does not exist on third-party marketplaces: in our study, we successfully posted our malicious apps using system package names on popular third party markets including `Amazon Appstore`, `appfutura.com`, `appslib.com`, `1mobile.com` and others. Also, the adversary can propagate the malware through other channels, for example, by sending a phone user the malicious app in email attachments or just a link to download the app.

Causes of the problem. All aforementioned Pileup flaws are caused by the conservative strategy Android has to take when updating a live system. After all, without a clear idea about what existing third-party apps and personal data are, it will not be wise to overwrite them and face uncertain liability consequences. In our opinion, that is why PMS chooses not to replace existing permissions, app data, packages with conflicting shared UIDs and even those bearing the names of new system packages[7]. This thinking will not go away even after those Pileup flaws are revealed. The oversight on the Android developer's side, however, is failing to connect the outcomes of this conservative strategy to the way Android refers to system capabilities, which is often by names, UID, etc. As a result, those critical security flaws have been left there for years, distributed across all AOSP versions as well as manufacturers' customizations.

Addressing the Pileup issues can be much more complicated than releasing yet another Android patch. This time, tens of thousands of Android source code versions need to be fixed and compiled, and billions of devices will be affected. Also, it is less clear to us whether Google and device manufacturers are willing to be more aggressive, taking back the conservative strategy and replacing unknown information assets on their customers' devices whenever they think is necessary. In our research, we explored an alternative, less extreme path to move forward, which automatically identifies potential hazards before an update happens (Section IV).

Responses from vendors. We notified Google of those Pileup vulnerabilities and provided them with all technical details soon after we confirmed the presence of the flaws within their Android versions. After analyzing our report, Google security team informed us that they came up with a fix for the permission bug (11242510: "Apps can hijack and change protection level of some permissions when Android is upgrading") and released it to their partners. They also created tracking numbers for all other issues that we reported and are working on solutions. We continue to communicate with them to find out the status of this system patching and are helping them fix all the Pileup vulnerabilities we discovered.

[7] Again, this can happen legitimately, for example, when one installs high-version Google Plus on Android 2.3.6.

IV. Finding Pileups

In this section, we elaborate the design and implementation of new tools for detecting Pileup flaws, called *SecUP*, which contributed to our finding of not only most flaws in Section III but also the scope and magnitude of the problem (existing on all AOSP versions and all the customized versions we studied, and affecting hundreds of new privileges and capabilities). Also, as discussed before, the root cause of the Pileup flaws is the conservative strategy used to update live systems, an issue that will not go away. To provide Android users timely protection without endangering their data, SecUP builds up a database that documents Pileup attack opportunities across thousands of Android versions, and operates a scanner app that leverages the database to detect Pileup risks on an Android device before its update.

A. Overview

Architecture. Figure 2 illustrates the architecture of SecUP, which includes a vulnerability detector, an exploit opportunity analyzer, a risk database and a scanner app. The detector verifies the source code of PMS (from different Android versions) to identify any violation of a set of security constraints, in which we expect that the attributes, properties (name, permission, UID, etc.) and data of a third-party app will not affect the installation and configurations of system apps during an update. A Pileup flaw is detected once any of those constraints are breached. All discovered vulnerabilities are further inspected by the analyzer, which searches released Android factory images for the opportunities of privilege escalations (depending on the new properties and packages on each Android version), and updates such information to a *Pileup risk database*. From such a database, the scanner app, which can be installed on Android devices, checks all installed third-party apps and reports to the user all the risks it finds. The app is designed in a way that minimizes false alarms particularly when the user has a high-version legitimate app on her device.

Example. To understand how SecUP works, consider the following example. As soon as a customized Android 4.3 is released, our SecUP service downloads its source code and manufacturer image. The Pileup detector then analyzes the Java code of PMS and, hypothetically, confirms the presence of the permission flaws in Section III-A. This discovery triggers the risk analyzer, which extracts all new system packages from the image to find out new permissions they define. All these permissions are vulnerable to the aforementioned harvesting and preempting exploits and therefore recorded in the risk database. Once the scanner app is downloaded and installed to an Android device, it first checks the manufacturer, model and version of the device and utilizes such information to search the database. If the device turns out to run a customized 4.2 that can be upgraded to the 4.3, the scanner inspects all manifest information to detect the packages that either declare or request the permissions retrieved from the database. Such packages, if unrelated to the device's manufacturer, are all reported to the device owner, together with the risks they pose. The owner will decide whether to uninstall them before upgrading her system.

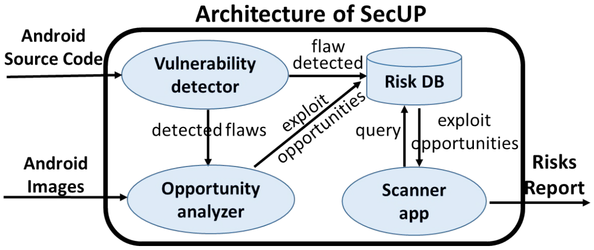


Fig. 2. Architecture of SecUP

B. Detecting Update Flaws

Pileup vulnerabilities are a type of program logic flaws that hide deeply inside PMS. To uncover those flaws in a mostly automatic way, SecUP includes a detector that performs a formal verification on the Java source code of PMS, in an attempt to identify the program logic that causes the problem. Here we elaborate this approach.

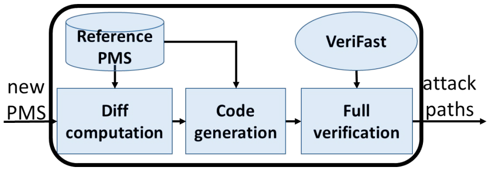


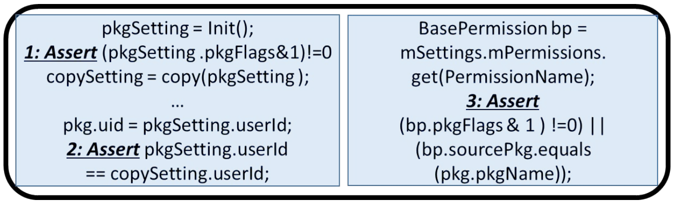
Fig. 3. Framework of Vulnerability Detector

Framework. As shown in Figure 3, our flaw detector takes as input the code of the PMS from an Android version serving as an upgrade target (called "new PMS") and outputs the execution paths that a malicious app can exploit to gain more privileges when the device is upgraded to this new OS. To this end, we want to convert the code into a format that can be automatically checked by an off-the-shelf Java verification tool, which in our implementation, is VeriFast [18]. VeriFast is a general-purpose verification tool for C and Java programs. It performs a full formal verification on the source code instead of bounded model checking [24]. As a result, it can guarantee completeness, i.e., finding all attack paths with respect to the assertions specified. However, just like other full verification tools [23], [21], [39], it requires the developer to manually annotate each function related to the assertions. This effort could be costly, depending on the scale and the complexity of the program [46]. It would be hard to verify a large number of Android OSes (including AOSP versions and vendor customized versions) in this way.

To make this verification process more automatic, we leverage the observation that changes to PMS during upgrading and customization have been minor and incremental. Specifically, we first built a *reference* PMS (AOSP 4.0.4), which was manually annotated with assertions and other descriptions (e.g., pre-conditions and post-conditions for a function, loop invariants) required for running VeriFast. This is a one-time effort, which facilitates the follow-up analysis on the PMS for other Android versions. Given an extracted new PMS, our detector first compares it with the reference to identify their code difference (*diff* for short). By inspecting the diff, it attempts to reuse the annotations in the reference when possible. Since oftentimes, the new PMS does not contain changes to the program locations that need annotations, though it often comes with some adjustments of the original program logic and therefore needs to be reverified, a new annotated source code ready for the verification can often be automatically created in this way. When the new PMS does bring in new program structures that need annotations (e.g., a new function), VeriFast will detect the changes it cannot handle and require manual adjustments during its runtime. This analysis framework is illustrated in Figure 3.

Assertions for Pileup detection. Most important to the verification is design of effective assertions for detecting the Pileup flaws. Those assertions are placed within PMS in the program logic related to updating to describe the security constraints that the program has to satisfy. Any instance of violating such constraints indicates a Pileup flaw. In general, two security constraints for PMS are: 1) a non-system app and its dynamic content should not gain any more privileges on the new OS than they have on the old Android it is upgraded from; 2) a non-system app should not compromise the integrity and the availability of the new Android (e.g., changing the settings and data of a system app). To describe these two constraints, we designed a set of assertions for two key stages during the installation of a new system package, that is, attribute setting and property registrations. As discussed before (Section II-A), for each new system package that comes with an update, PMS needs to configure its attribute settings (e.g., `userID`, `pkgFlags`, `SharedUserSetting`, `codePath`, `resourcePath`, etc.) and then register its properties (e.g., permissions, activities, services, receivers, content providers, permission trees, etc.). For each of these two tasks, we impose a set of assertions to make sure that the system app's attributes are not contaminated by a non-system app, nor can its properties be preempted by the app.

In the upgrading logic, the attribute settings for a new system app are filled according to the content of a class instance `pkgSetting`. The origin of the content, however, can be less clear to our flaw detector. The assertions for this process are designed to ensure that when content of `pkgSetting` is inherited from an existing non-system app on the old OS, individual fields in `pkgSetting` cannot be directly used to affect the settings of the system app being installed. To this end, our first assertion, placed right after initializing of `pkgSetting`, is evaluating whether the content comes from a system app. This is done by inspecting whether `pkgFlags` in `pkgSetting` has been set to 1 by PMS (note that this setting can be changed after the initialization). Whenever `pkgSetting` is used to configure individual attributes of the system package within its class `pkg` (parsed manifest information of the package), we further look at the content of `pkgSetting` to find out whether it has not been changed by the system since the initialization. Consider UID as an example. Immediately after `pkgSetting.userId` is assigned to `pkg.uid`, we insert an assertion claiming that `pkgSetting.userId` is equal to its value when `pkgSetting` is just initialized, as illustrated in Figure 4, where `copySetting` is a copy of the original value of `pkgSetting`. After analyzing the whole program, once VeriFast concludes that the first assertion is False (indicating `pkgSetting` possibly from a non-system source) while the second one for a specific attribute like UID is True (i.e., `userId` keeps the value from the non-system `pkgSetting`), we know that a Pileup flaw has been discovered. Intuitively, the evaluation results for this pair of assertions indicate that this specific attribute, which belongs to a new system app, can be set according to a non-system input.



```
pkgSetting = Init();
1: Assert (pkgSetting .pkgFlags&1)!=0
copySetting = copy(pkgSetting );
...
pkg.uid = pkgSetting.userId;
2: Assert pkgSetting.userId
== copySetting.userId;
```

```
BasePermission bp =
mSettings.mPermissions.
get(PermissionName);
3: Assert
(bp.pkgFlags & 1 ) !=0) | |
(bp.sourcePkg.equals
(pkg.pkgName));
```

Fig. 4. Assertions for PileUp Detection

For the registration logic, we want to ensure that whenever PMS stops registering a property of a system package, this is not caused by the presence of the same property owned by a non-system package with a different name. Otherwise, the existing property such as permissions can be elevated for a system use on the upgraded OS. Note that we here allow an existing third-party package to claim the property of a new system package when they have the same package name because in practice, a high-version legitimate system app installed on a low-version Android, is placed under `/data/app/` as a third-party app. Also, any logic flaws related to package names are already checked by the first two assertions. This constraint is expressed by the third assertion in Figure 4, which claims that either a conflicting old package is a system app (`pkgFlags` = 1) or the old package has the same name as the new system app. This assertion is inserted where the properties of the new system app are used to search `mSettings`. Whenever there is a hit (which stops the registration of the new app's corresponding properties), the assertion needs to be True if no logic flaw exists. Take permission as an example. Once the new permission name is discovered to be already existing in `mSetting`, we check $assert(((bp.pkgFlags\&1)! = 0)||(bp.sourcePkg.equals(pkg.pkgName)))$, where bp is a class instance storing information of the existing permission with same name. If the assertion is False, a flaw is reported.

Note that these two sets of assertions do not cover all possible Pileup vulnerabilities. For example, the first set cannot find a flaw when the UID of the new package has been changed since the initialization of `pkgSetting`, even though this change is still made based upon untrusted sources. As another example, the assertion for the registration logic misses the situation when the shared UID problem also exists: in this case, a malicious app can utilize the name of a new package to register its properties. Of course, this problem becomes less of a concern once the shared UID problem is discovered by the first set of assertions. On the other hand, whatever gets caught by those assertions is sure to be a Pileup flaw.

Other Android versions. As discussed before, the flaw detector is designed to efficiently generate annotations for a new PMS under an analysis. Given the source code of an Android version, it first checks code diff of PMS with the version's predecessor during an update. If no changes happen to PMS, then a verification is unnecessary if the predecessor's PMS has already been checked. Otherwise, the detector further extracts the code diff between the new PMS and the reference (or its immediate predecessor): as long as the annotated functions remain unchanged, all the marks can be moved to the new program for running the verification. When those functions have also been modified, what the detector does is to reuse as many existing annotations as possible and run `grep` to identify the new statements from the diff related to the variables in the assertions, and instrument those statements with the corresponding assertions. In this case, human intervention is needed to ensure that the code has been instrumented correctly.

C. Finding Exploit Opportunities

For a given Pileup vulnerability, what the adversary can get from it are pretty opportunistic, depending on the types of system attributes and properties involved in an update. With the fragmentation of the Android ecosystem, such exploit opportunities vary across not only different Android versions, but also different manufacturers, device models and sometimes even carriers. For example, Google and other vendors add to their devices different system apps, permissions, shared UID and others each time when updating their systems. On the other hand, our study shows that existing Android versions are all vulnerable (Section V). Therefore, it is important to understand the unique threats individual versions are exposed to. To this end, we built into SecUP an exploit opportunity analyzer (EOA for short) that automatically scans different Android versions to discover potential threats to their update mechanisms and also documents the problems identified in a well-structured Pileup risk database.

Android image scan. To find exploit opportunities in an update, EOA compares system attributes and properties on two consecutive Android versions from the same manufacturer, device model, region and carrier, based upon the Pileup flaws reported by the detector. As an example, with regard to all the flaws described in Section III, EOA inspects the OS image of Samsung GT-I9300-TGY on Android 4.1.1 customized by carrier TGY for new permissions, shared UID and package names not present in the immediate predecessor of the OS (Samsung GT-I9300-TGY Android 4.0.4). All such resources can be exploited during a Pileup attack and the security risks they expose need to be well documented.

To collect such information, EOA automatically identifies and downloads Android images from multiple sources, including all 38 Google Nexus images from [6] and 3,511 Samsung images from [16], to scan their pre-installed system apps. Those images are first classified according to their targeted device models, regions, carriers and manufacturers, and then arranged in the order of upgrade sequence (e.g., 2.3.6 to 4.0.4, then to 4.1.2). They are compressed files. To work on them, EOA first uncompresses each image to get `system.img`, which is the image of `/system/` directory for a running Android and contains all the system apps we need to look into. Then, EOA mounts the image using *ext4_utils* and inspects all the APKs under `/system/`. What it needs to look at depends on the types of Pileup flaws discovered by the detector. Again, for those we found (Section III), EOA focuses on the manifest files of individual APKs. After uncompressing the APKs using `apktool`, our approach extracts all defined permissions, shared UIDs, package names and other attributes and properties from their manifests, before unmounting the whole image and turning to the next one. In our research, we implemented the prototype of EOA using Python, which took about 700 hours to fully process all the 3,549 images from Google and Samsung.

Pileup risk database. The collected information from an Android version is kept in a data record. EOA further compares the records for two consecutive versions (according to the order of updates) to identify new resources vulnerable to the known Pileup flaws in the newer version's PMS. In the case of the flaws reported in the paper, we search for the permissions, permission trees, shared UIDs and package names on the newer version that are not there on the older one. All such attributes and properties can be taken advantage of by a malicious app running on the older Android and therefore are recorded into our Pileup risk databases, together with their meta data, including manufacturers, carriers, models, regions and others.

Note that this information collection process is not a one-time operation. Instead, we are closely monitoring the new versions (both official and customized ones) released by phone manufacturers and analyzing them whenever they become available so as to document the exploit opportunities they bring in.

D. Pileup Scanner

With the flaws identified by the detector and the exploit opportunities collected by EOA, we can provide a service to Android users to detect the privilege escalation attempts aimed at the system upgrading process. Such a service is delivered through a Pileup scanner app that operates on the user's device and checks the attributes and properties of all her third-party apps to find out those suspicious. This approach can offer timely and less intrusive protection than patching all vulnerable PMSes, given the fact that this will affect possibly billions of Android devices, all manufacturers and carriers. Also importantly, it is less clear to us how to fix the problem through patching without causing any collateral damages to the user's live system: actually even months after we notified Google and Android about those vulnerabilities, they still have not deployed effective solutions to address all the problems we discovered.

Following we elaborate the design and implementation of our Pileup scanning service.

Design. Our scanning service was built with three components: the Pileup risk database, a server and a scanner app. The server contains the flaw detector and EOA, and also a component to perform database operations and coordinate with the scanner app. To use this service, the Android user is supposed to download and install the scanner on her device. The app only asks for the INTERNET permission. Once it starts to run, it first gathers information about the device, for example, its brand, model and Android version from the class `android.os.Build`, and uses it to query the database for the security risks the device is exposed to. The server then responds with all the exploit opportunities the scanner needs to check, which for what we found, include new permissions to appear on the Android version the device will be upgraded to, new shared UIDs, package names etc.

The scanner app evaluates all third-party apps on the device against all those opportunities. Specifically, it first calls the API `getInstalledPackages` to get the names of a list of installed packages, and then uses `getPackageInfo` to retrieve all the information about these packages. What returned by the call includes a flag that indicates whether the package is a system one. Here we do not check system packages because all we want to prevent is that an unprivileged app escalates its privileges to the system level. From the API `getPackageInfo`, our app gets all the ingredients it needs for scanning a package, such as its name, signatures, permissions requested or defined, permission trees, shared UID and others.

Malware detection. Specifically, the scanner app first checks the permissions defined or requested by a third-party app. It reports a security issue if any permission defined by the app bears the same name as a new permission retrieved from the database and the app's package name is *different* from that of the system app on the new OS declaring the permission. This is because during an update, the new app will replace the existing one (from the old OS) of the same package name unless the former has a shared UID which is already taken by any malicious app on the old OS (The UID grabbing attack will also be detected by our scanner). Such a treatment is designed to avoid false positives, as the old system can have more recent system apps installed directory of under non-system apps: for example, Google Plus has been added when Android is upgraded from 2.3.6 to 4.0.4 on Google Nexus S phone; before the system update, one can actually install this system package on 2.3.6 as a third-party app, which further declares some legitimate permissions supposed to be defined at 4.0.4. Similarly, our app inspects the permission tree defined by the third-party app and raise an alarm whenever the tree is found to cover a single new permission declared on the next Android version, according to the information retrieved from the risk database. As an example, consider a permission `READ_CALL_LOG` defined by the system package `Android` when the system goes from 4.0.4 to 4.1.2 on a Nexus device. On 4.0.4, our scanner reports a security risk once it discovers that a third-party app with a different name also declares that permission. When this happens, the scanner informs the device user of the problem through an alert that explains to her the potential security consequences of keeping the suspicious app there during an update and suggests the actions she can take (e.g., uninstalling the app). This same strategy is also applied to handle the permission a third-party app requests: if it is a "future" permission and the app does not have a right name, the scanner alarms a risk and notifies the user that the permission will be automatically given to the app in an update.

For the shared UID taken by a third-party app, again, our scanner checks whether it is in conflict with the one used by a system app to be added in an update. If so and the third-party app carries a different package name than that system app, the scanner immediately reports a security risk. In the case that the app looks like that system app, with the same package name, the situation becomes a bit complicated, given the possibility that the system app of a higher version can be installed by the user before the update as a third-party app. To ensure that it is indeed the right app, the scanner further verifies the signature of the third-party app, which can be extracted through function `collectCertificates` of `PackageParser` class. If the app has not been signed by the same party that developed the system app (whose signature is stored in the risk database), an alarm is raised to make the user aware of this security risk. In the same way, the scanner handles the conflicting package name carried by existing and new apps. If the former does not have the right signature, we will report the problem.

V. Measurement and Evaluation

In this section, we report our study on the impacts of the Pileup risks to the Android ecosystem. We found that such security flaws are present in the source code of all AOSP versions and all 3,522 customized Android versions from Samsung, HTC and LG that we inspected, which strongly indicates their existence in all Android devices in the market. Our study further measured the distribution of exploit opportunities across different vendors, carriers and regions, bringing to light a few interesting findings. We further describe an evaluation of our scanner app's effectiveness in detecting Pileup malware and its performance.

A. Experiment Settings

Android image collection. To understand the scope and the magnitude of the problem, we collected a large number of Android OS images for our study. Specifically, we downloaded all 38 OS images for Google Nexus devices from [6], which cover all versions of Nexus 7, Nexus 10, Nexus Q, Galaxy Nexus, Nexus S from Android 2.3.6 to 4.3. We also gathered 3,511 OS images for Samsung devices from [16], covering 217 devices models and 267 carriers from Android 2.3 to 4.3. Those customized images were automatically crawled from the website [16], using paid premium accounts. We selected the images with consecutive version numbers so that the new resources added through updates can be identified. Those images took about 3 TB storage and 310 hours to download.

We also collected the source code of all AOSP versions and those customized by different manufacturers to detect the Pileup flaws within their PMSes. Such customized versions, including 1,552 from Samsung, 377 from LG and 1,593 from HTC, were downloaded from [15], [12], [9]. It took about 200 hours and 400 GB local storage.

Other settings. We further evaluated our scanner app on a Google Nexus S and a Galaxy Nexus phone. Our experiments were conducted with the OSes on those devices being continuously upgraded, starting from 2.3.6 for the Nexus S and from 4.0.4 for the Galaxy Nexus.

B. Pileup Flaws

Flaws detected. Table IV illustrates all the Pileup flaws we detected. As mentioned before, what motivated this research is our discovery of two permission Pileup flaws (Section III-A) and the first shared UID flaw (Section III-B), which was done manually. All other vulnerabilities, except the issue with Google Play Services (Section III-D), were all found automatically by our flaw detector (Section IV-B).

We further evaluated the detector on manually identified flaws. It worked on both the permission preemption and the shared UID vulnerabilities. However, it could not find the permission harvesting problem, that is, when a malicious app on a low version OS requests a permission available only on its successive version. This is because the flaw here is in the way

PMS registers non-system app's property, which our assertions do not cover. Similarly, our approach also could not detect the flaw associated with Google Play Services, as this service is installed under the `/data/app` directory on Android 4.0.4, and not considered to be an official system app. Design of assertions for user apps is challenging, given the difficulty in avoiding false positives (erroneously incriminating legitimate apps). How to address this issue and improve the coverage of the automatic flaw detection is left for future research.

Logic Flaw	Detected
Permission Preempting	Yes
Shared UID Grabbing	Yes
Data Contamination	Yes
Permission Tree	Yes
Other Flaws (pkg.mExtras)	Yes

TABLE IV. DETECTED PILEUP FLAWS

Impacts. In our study, we ran the detector against the PMSes within four recent AOSP OSes (4.0, 4.1, 4.2, 4.3) with 4.0 serving as the reference. It turned out that changes to other versions with regard to 4.0 are rather minor. As a result, most notations made on the reference were reused and the verification process was largely automatic. Our Pileup detector further inspected other customized Android versions. By comparing them with their corresponding AOSP versions, no changes were found within their PMSes[8]. Therefore, we conclude that all those versions contain the Pileup flaws discovered in our research[9]. Given that all 3,522 customized source code versions we studied have the same Pileup issues, we believe that this strongly indicates that the problem exists on all Android devices.

C. Measurement of Opportunities

The success of a privilege escalation attack on an update process depends not only on the presence of Pileup vulnerabilities, but also on the new system resources and capabilities the update adds that can be acquired by the adversary through the attack. Here we present a measurement study in which we ran our EOA (Section IV-C) against a large number of Android images to understand the exploit opportunities (new exploitable attributes and properties) they bring in.

Landscape. We first looked at the overall impacts of the Pileup vulnerabilities to the Android ecosystem, in terms of *update instance*, which refers to the upgrade of a specific OS (from a specific manufacturer, on a specific device model and for a specific carrier) to a higher one under the same set of constraints. For each update instance, we measured the quantity of exploit opportunities it can offer, with regards to all the Pileup flaws found in our research, such as the numbers of new permissions, packages and shared UIDs an update instance introduces to the new system. From the 38 Google and 3,511 Samsung images we downloaded, we identified 741 update instances. The statistics on their total exploit opportunities in each instance are illustrated in Figure 5. Particularly, we found that 50% of those instances have more than 71 opportunities.

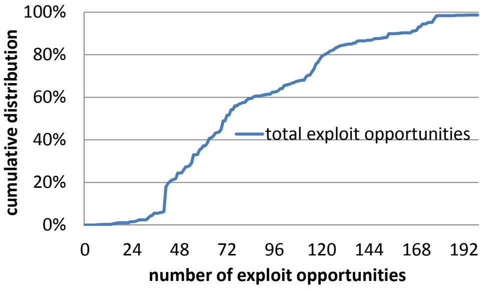


Fig. 5. Cumulative Distribution of Total Exploit Opportunities in Each Update Instance

Permissions, packages and UIDs. Then we inspected new permissions and packages. Here we call a permission *sensitive* if it is at least `dangerous` in the Android protection level, and *restrictive* if it is above the `dangerous` level. Figure 6 shows the cumulative distributions of the new permission number, sensitive and restrictive permission numbers respectively. Note that all the new permissions, including sensitive and restrictive ones, can be obtained by the malicious app exploiting the Pileup vulnerabilities and lowered down to the level of `normal`. Figure 6 demonstrates that 50% of the update instances offer at least 38 sensitive and at least 31 restrictive permissions each to the adversary.

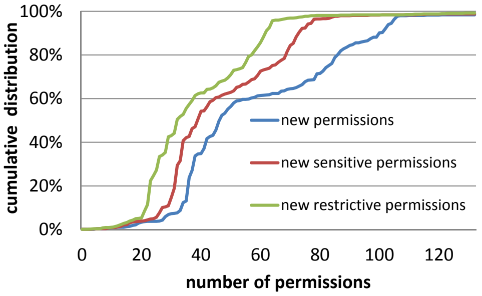


Fig. 6. Cumulative Distribution of New Permissions, New Sensitive and Restrictive Permissions in Each Update Instance

Also, updates render a lot of new packages and shared UIDs up for grabs. From the cumulative distributions in Figure 7, we can see that 50% of update instances have at least 23 new packages. Also at least one new shared UID was added in 50% update instances.

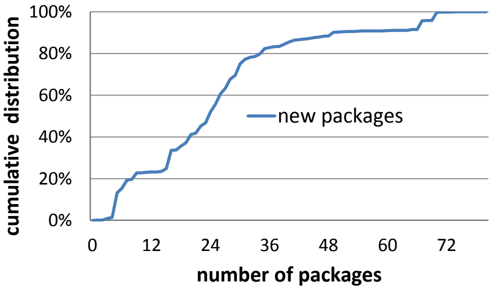


Fig. 7. Cumulative Distribution of New Packages in Each Update Instance

Impacts of customizations. Customizations of Android OSes contribute to a large portion of all the exploit opportunities discovered. Device manufacturers and carriers tend to add more permissions, packages and others than normal updates of AOSP versions. Figure 8 compares the average numbers of the exploit opportunities provided by AOSP, Google and

[8] Note that missing of the PMS in the source code of a customized Android indicates that the customization makes no changes to the PMS. This is because in order to build an OS image, one needs to copy the source code of the customized Android onto that of AOSP, overwriting the files to be replaced, before compiling the code.

[9] This does not include the two flaws our approach missed.

Samsung, when the system is upgraded from 2.3.X to 4.0.X, then to 4.1.X, 4.2.X, 4.3.X and 4.4.X consecutively. As we can see from the figures, not only do the manufacturers introduce more opportunities than AOSP, but Samsung adds more than Google. Also interestingly, though Google and AOSP make the biggest system overhaul from 2.3.X to 4.0.X and show a trend of less aggressive updating afterwards, Samsung continues to bring in more new stuffs from 4.1.X to 4.2.X and to 4.3.X, at the cost of increased security risks.

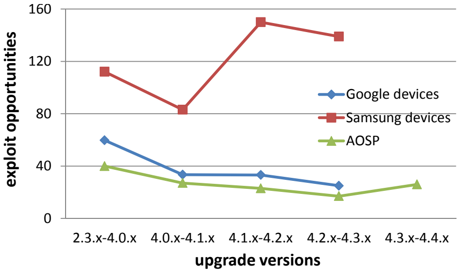


Fig. 8. Exploit Opportunities Affected by Vendor Customization

We also studied the diversity and complexity different carriers bring to the picture. We find that they have significant influence on the exploit opportunities within their customers' systems. Table V presents what we found from the carriers in eight different countries, based upon the 3,511 images downloaded from [16]. From the table, we observe that each carrier is associated with many new functionalities for each update, which makes their devices susceptible to the Pileup exploits. Particularly, when the system goes up from 4.0 to 4.1, the devices affiliated with DCM in Japan and TMB (T-Mobile) in US look most vulnerable. For the 4.2 update, the devices with DBT in Germany, INU in India and SER in Russia offer the adversary over 120 opportunities each.

Update Versions	**Average Exploit Opportunities per Update Instance by Carriers**							
	CHU (CN)	DBT (DE)	DCM (JP)	FTM (FR)	INU (IN)	SER (RU)	SKT (KR)	TMB (US)
2.x -4.0		108	103	110	102	115	72	98
4.0 -4.1	69	75	137	32	95	78	72	162
4.1 -4.2		129			193	131		

TABLE V. INFLUENCE OF CARRIERS ON EXPLOIT OPPORTUNITIES

D. Evaluating Scanner

Effectiveness. To evaluate the effectiveness of the scanner, we first set up an Android 2.3.6 on our Nexus S and a 4.0.4 on the Galaxy Nexus. Under these OSes, we installed top 100 free apps from Google Play and a set of attack apps for exploiting all the flaws in Section III. Also on these devices we installed system apps that could be updated through Google Play, i.e. Google Play Services, in order to better evaluate potential false positives. Before the systems were upgraded, we ran our scanner on them to detect potential Pileup risks. After that, we changed our malicious apps according to the exploit opportunities on the new systems, scanned the systems again and continued to update them to the next versions, until 4.3, the most recent one available. During those updates, our scanner successfully detected all malicious apps and never incriminated any legitimate ones, including the high-version apps.

Performance. We measured the performance of our scanner app, in terms of the amount of the time it used to scan apps before each update. This includes what it took to check the apps on the devices(`Local Time`) and what was introduced by network communication(`Query Time`). Such a delay(`Total Scan Time`) was further compared with the durations of the upgrading on those devices. Table VI presents the experimental results for scanning 30 apps on Nexus S and 100 apps on Galaxy Nexus. Note that 30 apps are approaching the limit Nexus S can handle smoothly. Our study shows that the time spent on checking Pileup risks only took a negligible portion of that incurred by the whole upgrading process.

Phone Model	Nexus S		Galaxy Nexus		
Update Version	2.3 -4.0	4.0 -4.1	4.0 -4.1	4.1 -4.2	4.2 -4.3
Local Time(s)	0.318	0.446	1.284	0.44	0.374
Query Time(s)	2.484	1.532	1.216	0.812	0.446
Total Scan Time(s)	2.802	1.978	2.5	1.252	0.82
Upgrading Time(s)	406	546	657	708	745
Scan/ Upgrading	0.69%	0.36%	0.38%	0.18%	0.11%

TABLE VI. PERFORMANCE EVALUATION OF SCANNER APP

VI. DISCUSSION

Other Pileup flaws. Our current design to detect Pileup flaws is very preliminary. We only checked part of the program logic within PMS for setting a new system package's attributes and registering its properties. This is far from sufficient, given the Pileup flaws that may also appear in other part of the code. A prominent example is that related to the installation of non-system apps. As discussed before, our detector does not work on the permission harvesting exploit and the denial-of-service attack on Google Play Service, because in both cases, the problem is within the program logic for processing non-system apps. Even for system apps, the assertions used in our implementation are just a set of sufficient but unnecessary conditions for the presence of Pileup flaws. As a result, there is no guarantee that we found all such problems even within the code related to system apps' attributes and properties.

Another issue is how to make SecUP more automatic. This is challenging, given the difficulties in detecting such logic flaws even with our semi-automatic techniques. A more capable solution can be built on more mature program verification tools that less depend on annotations, and a more powerful program analysis technique that automatically generates the annotations and instrument a program correctly.

Other services and OSes. Our study only focuses on PMS, which is just one service involved in the upgrade process. There are many other services and components, such as UserManagerService, BackupManagerService, DevicePolicyManagerService and ServiceManager, etc., that can also have various types of Pileup vulnerabilities. More importantly, given the fundamentality of the issue, we suspect that a similar problem could also exist in other mobile systems. Further research is needed here to better understand the scope and the magnitude of this new security hazard.

VII. RELATED WORK

Security issues in upgrades. Updates (or patches) have long been used to quickly respond to software vulnerabilities discovered [28] and mitigate the threat of exploiting such flaws. Although it is known that patches themselves may bring in new flaws and releases of updates also tips the adversary about the vulnerabilities they are meant to protect [26], [19], never before has anyone systematically studied the security hazards introduced by the vulnerable program logic for installing such updates. Understanding this issue is particularly important to securing mobile devices, whose OSes need to be upgraded frequently, in the presence of a large amount of critical user data and applications. The program logic for such a live update inevitably becomes more complicated and thus more error-prone. Our research makes a first step towards better understanding of this new security challenge and finding a practical way to address it.

Most related to our is the recent research on dormant permissions [43], which is published right before the submission of this paper and thus just comes to our awareness. The research focuses on Android permission management and evolution but also reports an experiment in which an app was made to apply for a new dangerous permission or the one defined by the app yet to install and waited until the installation of the right app or an upgrade to get it. This is similar to the permission harvesting threat we found. However, the research did not investigate the updating mechanism at all and thus failed to identify the root cause of the problem and any other vulnerabilities, which are often much more serious. Particularly, permission harvesting only grants the malicious app dangerous permissions, while our permission preemption attack allows a malicious app to acquire *any* permissions, including system and signature level ones, and even lower down their protection levels. Also after the malware performing such preemptions is detected, removing it is nontrivial, because it will go with the permission, which denies other apps' access to the protected resources. Beyond permissions, our research brought to light other serious logic flaws: e.g., the shared UID issue that allows the installation of a malicious calendar app and the data contamination threat that causes cross-site scripting, login CSRF, etc. Most importantly, we systematically studied PMS, found the root cause of the problems, analyzed over 3,000 images to understand the gravity of those logic flaws and further offered a mitigation.

Mobile OS security. Our work on the Pileup flaws falls into recent efforts on discovering and mitigating new security threats to mobile computing systems, a vibrant research area. Numerous studies have been done to exploit the implementation errors in mobile apps [32], [22] as well as circumvent Android's sandbox and permission protection [29], [34], [36], [41], [42]. Examples include the Permission Re-Delegation attack [34] and other types of confused deputy problems for the mobile platforms [44]. Also, new security designs have continuously been proposed to address those challenges and enhance Android security mechanisms [44], [31], [35], [30]. Compared with such prior research, our study explores mobile security from a different angle: our malicious app does not aim at the vulnerabilities within the "current" OS on which it is installed, but rather the "future" system the OS will be upgraded to. This new way to look at the problem helps broaden the scope of security research in the area.

Flaw detection. Formal reasoning tools have been widely used for bug findings [20], [27], [25], [33]. In our research, we built our vulnerability detector upon a popular formal reasoning tool for Java (i.e., VeriFast[38]), which can certainly be replaced by other similar tools. Also, the way our scanner app checks third-party packages is essentially signature-based malware detection, which can potentially help improve the existing antivirus systems like Norton Security [10] to handle the malicious code exploiting Pileup vulnerabilities.

VIII. Conclusion

Android devices are frequently upgraded, replacing and adding tens of thousands of files on a live system in the presence of a large amount of user data and existing apps. To ensure that this process goes smoothly without endangering such user assets, the Android update mechanism involves complicated program logic and inevitably becomes error-prone. In this paper, we report the first systematic study on the security implications of this problem. Our research reveals Pileup, a new type of privilege escalation vulnerabilities within the updating logic. Exploiting Pileup flaws, a malicious app can use what it declares on a low-version system to gain system capabilities on the new OS after an upgrade, involving gaining system and signature level permissions, substituting system apps, contaminating browser data and blocking the installation of new system apps. We performed a large-scale measurement study to confirm the presence of such flaws in all Android versions, official or customized. To mitigate the threat they pose, we further developed SecUP, a new service that detects Pileup vulnerabilities from released system code, automatically gathers attack opportunities and leverages such information to support a scanner app running on the user's device to identify the malicious code attempting to exploit Pileup flaws.

Acknowledgements

This work is supported in part by the NSF CNS-1017782, 1117106, 1223477 and 1223495. We also thank Shaz Qadeer for his help on VeriFast.

References

[1] Android Developers. http://source.android.com/.
[2] Android Distribution. http://www.droid-life.com/tag/distribution/.
[3] Android Fragmentation. http://opensignal.com/reports/fragmentation-2013/.
[4] Android Permission. http://developer.android.com/reference/android/R.styleable.html#AndroidManifestPermission_protectionLevel.
[5] Android Version History. http://en.wikipedia.org/wiki/Android_version_history.
[6] Factory Images for Nexus Devices. https://developers.google.com/android/nexus/images.
[7] Google Cloud Messaging. http://developer.android.com/reference/com/google/android/gms/gcm/GoogleCloudMessaging.html.
[8] Google Play Services. https://play.google.com/store/apps/details?id=com.google.android.gms.
[9] HTCdev. http://www.htcdev.com/devcenter/downloads/P00.
[10] Norton Security Antivirus. https://play.google.com/store/apps/details?id=com.symantec.mobilesecurity&hl=en.
[11] One Billion Android Devices. http://www.technologyreview.com/graphiti/520491/mobile-makeover/.

[12] OpenSouce Code Distribution. http://www.lg.com/global/support/opensource/index.

[13] Permission Tree. http://developer.android.com/guide/topics/manifest/permission-tree-element.html.

[14] PileUp Supporting Materials. https://sites.google.com/site/pileupieeesp/.

[15] Samsung Open Source. http://opensource.samsung.com/.

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

[17] Shared UID. http://developer.android.com/guide/topics/manifest/manifest-element.html#uid.

[18] VeriFast. http://people.cs.kuleuven.be/~bart.jacobs/verifast/.

[19] A. Arora, R. Krishnan, A. Nandkumar, R. Telang, and Y. Yang. Impact of vulnerability disclosure and patch availability-an empirical analysis. In *Third Workshop on the Economics of Information Security*, 2004.

[20] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. *ACM SIGOPS Operating Systems Review*, 2006.

[21] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal methods for Components and Objects*. Springer, 2006.

[22] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *the 17th ACM conference on Computer and communications security*, CCS '10. ACM, 2010.

[23] B. Beckert, R. Hähnle, and P. H. Schmitt. *Verification of object-oriented software: The KeY approach*. Springer-Verlag, 2007.

[24] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in computers*, 2003.

[25] P. Bjesse, T. Leonard, and A. Mokkedem. Finding bugs in an alpha microprocessor using satisfiability solvers. In *Computer Aided Verification*, 2001.

[26] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *IEEE Symposium on Security and Privacy*, 2008.

[27] C. Csallner and Y. Smaragdakis. Check'n'crash: combining static checking and testing. In *the 27th international conference on Software engineering*. ACM, 2005.

[28] W. Cui, M. Peinado, H. J. Wang, and M. E. Locasto. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *Security and Privacy, IEEE*, 2007.

[29] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on android. In *Information Security*. Springer, 2011.

[30] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *USENIX Security Symposium*, 2011.

[31] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, 2010.

[32] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *USENIX Security Symposium*, 2011.

[33] D. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In *Verification, Model Checking, and Abstract Interpretation*. Springer, 2004.

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

[35] P. Gilbert, B.-G. Chun, L. P. Cox, and J. Jung. Vision: automated security validation of mobile apps at app markets. In *the second international workshop on Mobile cloud computing and services*. ACM, 2011.

[36] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *the 19th Annual Symposium on Network and Distributed System Security*, 2012.

[37] M. Howard and S. Lipner. Inside the windows security push. *Security & Privacy, IEEE*, 2003.

[38] B. Jacobs and F. Piessens. The verifast program verifier. *CW Reports*, 2008.

[39] T. Lev-Ami and M. Sagiv. Tvla: A system for implementing static analyses. In *Static Analysis*, pages 280–301. Springer, 2000.

[40] J. Oh. Fight against 1-day exploits: Diffing binaries vs anti-diffing binaries. In *Blackhat Technical Security Conference*, 2009.

[41] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *the 18th Annual Symposium on Network and Distributed System Security*, 2011.

[42] S. Schrittwieser, P. Frühwirt, P. Kieseberg, M. Leithner, M. Mulazzani, M. Huber, and E. Weippl. Guess whos texting you? evaluating the security of smartphone messaging applications. In *the 19th Annual Symposium on Network and Distributed System Security*, 2012.

[43] J. Sellwood and J. Crampton. Sleeping android: Exploit through dormant permission requests. In *3rd Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2013.

[44] R. Wang, L. Xing, X. Wang, and S. Chen. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *the 20th ACM conference on Computer and communications security*. ACM, 2013.

[45] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *USENIX Security Symposium*, 2002.

[46] J. Yang and C. Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *ACM Sigplan Notices*, 2010.