Life after App Uninstallation: Are the Data Still Alive? Data Residue Attacks on Android

Xiao Zhang
Syracuse University
xzhang35@syr.edu

Kailiang Ying
Syracuse University
kying@syr.edu

Yousra Aafer
Syracuse University
yaafer@syr.edu

Zhenshen Qiu
Syracuse University
zqiu02@syr.edu

Wenliang Du
Syracuse University
wedu@syr.edu

***Abstract*—Uninstalling apps from mobile devices is among the most common user practices on smartphones. It may sound trivial, but the entire process involves multiple system components coordinating to remove the data belonging to the uninstalled app. Despite its frequency and complexity, little has been done to understand the security risks in the app's uninstallation process. In this project, we have conducted the first systematic analysis of Android's data cleanup mechanism during the app's uninstallation process. Our analysis reveals that data residues are pervasive in the system after apps are uninstalled. For each identified data residue instance, we have formulated hypotheses and designed experiments to see whether it can be exploited to compromise the system security. The results are surprising: we have found 12 instances of vulnerabilities caused by data residues. By exploiting them, adversaries can steal user's online-account credentials, access other app's private data, escalate privileges, eavesdrop on user's keystrokes, etc. We call these attacks the data residue attacks.**

To evaluate the real-world impact of the attacks, we have conducted an analysis on the top 100 apps in each of the 27 categories from GooglePlay. The result shows that a large portion of the apps can be the target of the data residue attacks. We have further evaluated the effectiveness of popular app markets (GooglePlay, Amazon appstore and Samsung appstore) in preventing our attacking apps from reaching their markets. Moreover, we have studied the data residue attacks on 10 devices from different vendors to see how vendor customization can affect our attacks. Google has acknowledged all our findings, and is working with us to get the problems fixed.

I. Introduction

The popularity of Android continues, with more than one billion accumulated device activations and 81.5% market share as of 2014, according to the report from IDC [8]. In the same year, GooglePlay, the official Android market, reached 1.3 million applications (apps, in short) and more than 50 billion downloads [6], [13]. However, the real app engagement is surprisingly low. A recent study by Localytics in 2014 indicates that 20% of apps are used only once [10]. In addition, a report from iResearch on China's smartphone market shows that 85% of users delete downloaded apps from their devices within one month, and after 5 months, only 5% of apps remain [9]. The short lifespan of apps is caused by many reasons, such as annoying notifications, buggy UI, complex registration processes, etc. Privacy is also a major cause. For example, the stand-alone Facebook Messenger app requests a scary long list of permissions, and according to a poll by AndroidCentral, which was conducted on more than 7,700 people, nearly one third of users uninstalled the app because of privacy concerns [4].

These reasons lead to frequent app uninstallation. An important security question is what will happen if an app is uninstalled, but its data are not completely cleaned from the system. This question may not be a major concern on the traditional computing platforms, because when an app is uninstalled, its data still belong to the users, and the security parameters of those data do not change. In Android, when an app is installed, except in some special situations, a new user is created. The app will be executed using this new user's privileges. When an app is uninstalled, the user will be deleted. Any data left behind by this app now become "orphans", because their owner no longer exists. They may not do any harm if they stay as "orphans". However, if they are inherited or possessed by another app, i.e., another user, there will be potential security consequences if the "orphan" knows a lot about its previous owner or still possesses some privileges of the previous owner. We call the problem caused by these "orphans" the data residue problem.

The Data Residue Problem The data residue vulnerability is particularly complicated due to the fact that the residue might take several forms. During runtime, the system may store various types of data on behalf of apps, ranging from app permissions, operation history, user configuration choices, etc. These data can be files, databases, and in-memory data structures. They may not be simply data; they can represent privileges (such as capabilities), i.e., whoever possesses them can gain additional power. For example, the URI placed on Android Clipboard by an app gives recipients the capability to access that app's private data.

Android has made reasonable efforts to clean up the data owned by an app during the uninstallation process. However, given the sheer complexity of the interaction between apps and the system, which leads to the wide scattering of app data inside the system, it is very challenging to do a complete job. Due to these reasons, data residues become very common in Android. However, having data residues does not necessarily lead to security problems. It remains unclear whether Android's existing defense mechanisms and system design are

NDSS '16, 21-24 February 2016, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2016.23061

robust enough to mitigate the security breach caused by data residues.

Based on the nature of data residues, we came up with several interesting hypotheses and questions: (1) Most data in Android are protected by user ID, so what will happen if the user ID belonging to the uninstalled app is given to a new app? (2) What if the system intentionally or unintentionally gives a data residue to another app? (3) What are potential problems if a data residue is a capability? (4) What are the conditions that can make it possible for another app to gain the ownership of a data residue? Intrigued by these questions and motivated by some encouraging preliminary discoveries, we launched a systematic study of the data residue problem in Android.

Our methodology and findings Data residue can happen in several places, but the residue instances inside Android's system services have the highest risk, because developers and users are hardly aware of their existence, and these services are privileged. We have analyzed 122 system services in Android Open Source Project (AOSP) codebase 5.0.1. The analysis is difficult to be fully automated because it depends on significant amount of domain knowledge about each specific system service. Although data residues caused by the lack of data removal logic are relatively easy to detect, several identified cases are caused by flawed code logic, the detection of which requires system-level code understanding and/or sophisticated experiment design for exploitation. Our ultimate goal would be developing an automatic detection system to eliminate all data residue instances from the Android system. In this project, however, we take the first step towards understanding the severity of this issue. Therefore, we manually inspected the source code of Android system services, formulated hypothetical attacks, and then designed experiments to verify whether the attacks would work or not. Upon failed attempts, we further examined the reasons behind. This entire process took six person months to finish.

Our investigation results indicate that the data residue problem in Android is truly worrisome. From the 122 system services, we have found 12 data residue instances that can lead to attacks. The data in each instance serves different yet security-critical purposes, empowering adversaries to subvert Android's built-in protections. For example, we found that if an app uses Android's credential management services, such as `AccountManager` or `Keystore`, the credentials for the user's online accounts can become data residues after the app is uninstalled. We designed an experiment to show that a malicious app can "inherit" these credentials and therefore completely take over the user's online accounts. Many apps use `AccountManager`. For example, `myMail` is a popular email client app, with one million downloads. It uses `AccountManager` to store the credentials for all the email accounts it manages, including Microsoft Exchange, Gmail, and Yahoo Mail. After this app is uninstalled, our malicious app can take over all its credentials stored inside `AccountManager`, and can successfully log into the user's Yahoo Mail, Gmail, and Microsoft Exchange accounts.

Our research also reveals that data residues can lead to privilege escalation. For example, an app can leave a maliciously crafted reference (a form of granted privilege) on Android Clipboard to allow others to access its internal resource. When the app is uninstalled, the reference becomes useless, because the targeted resource is not there anymore. However, the reference is still kept on Clipboard, and hence becomes a data residue. When a victim app is installed, if its protected resource matches with the reference crafted by the uninstalled malicious app, any app on the device can now use the data residue to access the protected resource inside the victim app. In our experiments, we have successfully gained the access to the user's mailbox in `Yahoo Mail`, files stored in `OneDrive`, and bank statements inside the official `Chase` app. The credential residues from `AccountManager` and URI residues from Clipboard are only two cases in our discoveries. In this paper, we will present the details of all the vulnerabilities and attacks discovered in our research.

To further understand the feasibility of our attacks in the real world, we tried to upload our attack apps (without causing real damage) to different Android markets, including GooglePlay, Amazon and Samsung appstore. We would like to see whether these markets have adequate defense mechanisms to make our attacks infeasible. Our results show that most of our attack apps, with some exceptions, can actually be published in those stores, indicating that our data residue attacks have real impact. Moreover, we tried all our attacks on 10 devices from different vendors running different versions of Android. The high success rate of these attacks indicates that the device customization made by vendors does not make their devices more resilient against our data residue attacks.

Google Response As millions of users are at risk because of the vulnerabilities discovered in our study, we tried our best to keep the issue confidential. For each identified data residue attack, we have submitted a detailed report to Google, along with illustrative videos to demonstrate the attacks and damages. Google has acknowledged all our findings and labeled 7 of them as medium-priority vulnerabilities. In the meantime, we are working closely with Google to fix all issues. The status update for each vulnerability, as well as demonstration videos and analysis results on real apps, are available at the following anonymous website [7].

Contributions The contribution of our work is three-fold:

- We have discovered a class of vulnerability, i.e., data residue vulnerability, in the Android system. We have successfully developed attacks to exploit these vulnerabilities.
- We have conducted a systematic investigation of the data residue vulnerability on all system services in Android. Our methodology can be adopted by the developers of the Android OS to improve its resilience against the data residue attacks.
- We have also performed a thorough evaluation on the potential damages of the data residue problem in the real world.

Roadmap The rest of this paper is organized as follows: Section II explains the necessary background knowledge and then formulates the data residue problem on Android. Section III describes the methodology used in our systematic investigation. Section IV shows the discovered data residue vulnerabilities and the actual attacks. Section V systematically evaluates the damage scope from three different perspectives.

Section VI discusses the fundamental causes and potential defense approaches. Finally, Section VII describes the related work and Section VIII makes conclusions.

II. Problem Formulation

A. Background

The lifecycle of an app on Android devices can be divided into three stages: installation, interaction and uninstallation. This section provides a further explanation on each stage.

Installation For security reasons, Android isolates apps from one another and from the system by assigning them a distinct Linux User ID (UID) during the installation process. The UID does not change for the duration of the app's lifetime on the device. The system maintains a list of UIDs in use, and assigns the next available one to the newly installed app. Device rebooting will force the system to reconstruct the UID list, so the UIDs of the uninstalled apps will be recycled and be possibly assigned to the newly installed apps. Android also creates a private folder for each app in the internal storage, and since Android 4.4, each app also gets an app-specific region on the external storage. Android does not require any permission for an app to access its own directories, but it does require permissions for sensitive resources. Framework level resources are granted via filling in the UID to permission map for this app, while hardware related resources, like Internet, Bluetooth and SDCard, are guarded by validating app's Group ID (GID). Granted permissions enables apps to conduct out-of-sandbox communication.

Interaction Apps frequently interact with the system and other apps during the runtime. Such interactions fulfill the necessity of resource sharing and functional cooperation. Most of the interactions are managed by Android's privileged services, which expose the low-level functions of the system (both Android framework and kernel) to the high-level apps. It should be noted that, even though most of the privileged services belong to the `system_server` process, some are provided by the privileged apps pre-installed in the system partition. In this paper, unless otherwise specified, we use system services to refer to the services of both types.

Interactions with system services come with a side effect: the Android framework actively stores app data inside the system in a variety of forms with or without app's awareness. For instance, the `Clipboard` service stores apps' clip data in memory, while the `AccountManager` service uses a database to save user credentials. In these cases, the data stored by the system services are still owned by and accessible to the requesting app, which is fully aware of the whereabout of the data. However, in many situations, apps' data are stored in system services without apps' awareness; these are mainly for caching and management purposes. For example, `PrintService` stores the failed printing jobs in a database. Although it does that for the benefit of apps, most apps do not know that their private data are stored somewhere else.

The extensive interaction with the system services results in app's data (private or public) being scattered throughout the system. This makes data cleanup extremely difficult when an app is being uninstalled. These data are actually well protected by Android's access control system when the app is still on the device, but after it is uninstalled, it is not well understood what can happen to these data if they are left on the device. As shown by our research, Android made many mistakes in dealing with data residues.

Uninstallation Uninstallation requests, which can only be initiated by the user of the device, are handled by the `PackageManager` Service (PMS). PMS first tries to kill the target app's process and notifies all the parties that are still communicating with this app via Android Binder's "link to death" facility. PMS then deletes all the app's private folders, including the one on the external storage. Files placed inside the shared folder on the external storage will not be removed (we do not consider these data as residues, because they are kept by design). Finally, PMS recycles the UID belonging to the uninstalled app, but does not reuse it until device rebooting.

Android has two main mechanisms to inform all parties in the system about the app uninstallation. The first mechanism is broadcast. After an app is uninstalled, PMS sends out a broadcast notification to the entire system; any entity can register for such a broadcast, and take actions upon receiving it. The second mechanism is called `PackageMonitor`, which monitors the status of the packages in the system. System services can use it to trigger their reactions when an app's installation status is changed. Both mechanisms can be used by system services to clean up data residue, but they are not widely used, causing many data residues in the system.

B. The Data Residue Problem

Given the fact that apps usually have sensitive data stored in scattered places inside the system, it is of paramount importance to notify all corresponding entities for data cleanup upon app uninstallation. Android strives to provide such a guarantee by deleting app's private folders when an app is uninstalled, but this is only the easy part; the challenging part is the data stored in system services.

What can go wrong Many things can go wrong in dealing with data residues. First, as the residue removal logic is not mandatory in the design of system services, not all system services take the responsibility to remove data when an app is uninstalled. For example, `DownloadService` is not even aware of the app uninstallation, because it does not register any handler to listen to the uninstallation event. Second, some services do try to delete data residues, but fail to do a complete job. For example, `PrintService` does react to the uninstallation event, but it does not clean up the failed-printing records made by the uninstalled app. Third, some system services try to find a new owner for data residues, without understanding the potential security consequences.

What makes the situation even worse is that multiple parties can jointly create data residues, and it is unclear who should take the responsibility to remove them during the app uninstallation process. For example, when users need to select a printing app to handle the printing job on the device, they trigger the Settings app, which sends a request to `PrintService` for the configuration update. In this case, three parties are involved: user, the Settings app and `PrintService`, but when the printing app is uninstalled, nobody takes the responsibility to remove the configuration

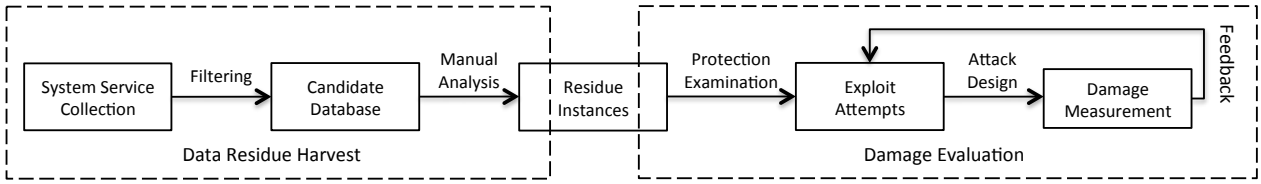

Fig. 1: Methodology of Data Residue Study on Android System Services

entry, which now becomes a residue. This latter residue allows a newly installed app with the same package name to become the device's default printing app, without user's approval. Another similar residue instance comes from `TextService`, which allows malicious apps to monitor user's keystrokes.

Android's UID-permission security architecture prevents unauthorized access to the data saved in system services, but no study has provided a thorough understanding on whether such a protection is still effective after the data's owner is uninstalled. We would like to fill this void by performing a systematic study to reveal the data residue instances in Android and understand their security consequences. We exclude the data intentionally left on devices, such as app's backup data and files on the shared external storage.

Assumption To conduct each attack, we assume the presence of a malicious app installed on the victim's Android devices. These apps do not need special privileges. Actually, in all the cases that we have discovered, the malicious app only needs a subset of the target app's permissions to perform the attack.

III. Methodology

We conducted our study in two phases: data residue harvest and damage evaluation. Figure 1 depicts the flow of our methodology.

Data Residue Harvest To uncover data residues, we look at two types of services, i.e., system services and the services in pre-installed apps, because both of them are privileged. We collect all available system services using the *dumpsys* utility provided by the Android Debug Bridge (*adb*). At the same time, we collect pre-installed apps's services by parsing their manifest files. We only focus on the services that are declared as exposed (private services are not accessible to other apps).

We manually analyze the source code of these services to identify data residues. Though static analysis seems like an alternative solution, the existing tools [3], [16], [42], [44] mostly stay at the app level and emulate system behaviors based on extensive domain knowledge on Android framework. In our analysis, the focus is on system behaviors, not on apps. Each system service behaves differently and requires its own domain knowledge, and manual code inspection seems like the best option to gain this knowledge. Moreover, a significant percentage of system services are written using a mixture of Java and C++ code, making automatic analysis even more difficult. Similarly, dynamic analysis [27], [37] does not fit our need either, as we have to manually identify all events that could trigger data residue and the conditions could be a combination of flaws from multiple system services. Given the small number of system services that we have to study, manual analysis turns out to be a more viable approach.

Our manual analysis is conducted based on the following two insights. First, we have observed that system services are meant for serving multiple apps, so the data collected from each app are clearly organized based on the owner app. This is also necessary for protection, so one app cannot use the data from another app. Files, database, and well-marked data structures (e.g. `Hashmap`) are used to store app-specific data. Using this clue, we focus on these data structures and File APIs (which also cover database accesses). Second, the awareness of app uninstallation is another clue. If a service is unaware of app uninstallation, any saved data naturally become residue. We also systematically examine corner situations that may subvert data cleanup logic, like the `AccountManager` case in Section IV-A.

Damage Evaluation Having data residues does not necessarily lead to security breaches, as long as the data are well guarded and the protection is still effective even after the owner is uninstalled. Though such a lifetime protection is theoretically feasible, Android seems to be confused in identifying the rightful owner of the residues. The main cause of the confusion is the implicit assumptions that Android made in its design. One of such assumptions made by system services is that app's identities are unique; so two entities with the same identity (e.g. UID or package name) should belong to the same app. It turns out that this assumption does not hold when the state of the device changes. Our study attempts to unveil these implicit assumptions and more importantly examine their validity. We consider three operations that can lead to device state changes: device reboot, app installation and app uninstallation. We create scenarios to make those assumptions false, and see how Android handles the data residues in these conditions.

Once a data residue instance is found to be exploitable, we conduct real-world attacks to measure all possible damages. The design of each attack builds upon the architecture derived from a comprehensive list of data operations. Inspired by the read, write and execute permissions on the traditional UNIX file system, we naturally test the accessibility, modifiability and utilizability on each instance. One notable insight is that, by the time of the exploit, the data owner has been uninstalled already, thus, malware will be less interested in altering the data content. However, it is of great importance to evaluate whether the data residue, which was initially associated with the uninstalled app, can be re-associated to another app.

IV. Attacks

We conducted our study on Android Lollipop (version 5.0.1) with a collection of 122 candidate service samples, including 96 system services and 26 public system-app services. The entire examination process (which took 6 person months) includes data residue harvest and damage evaluation. Table I

Samples (# Total/Candidate/Residue)	Category	Service Instances	Residues	Exploitable
System Services (96/96/10) System-app Services (161/26/2)	Credential Residue	AccountManager	User Credentials	✓
		Keystore	Public/Private Keypairs	✓†
	Capability Residue	Clipboard	URI	✓
		ActivityManager	PendingIntent	✗
	Settings Residue	TextService	User Selected Components	✓
		DebugService		✓
		DreamService		✓
		TrustAgent		✓
		LocationManager		✓
	History Residue	PrintService	Print/Download Information	✓
		DownloadService		✓†
	Permission Residue	PackageManager	Permissions	✓

† Resolved on Android Lollipop, but reproducible on KitKat and prior versions

TABLE I: Worrisome Data Residue Situation on Android System Services

summaries the study results. We are able to identify 12 data residue instances, which account for 10% of the candidate services. Technically, two of these 12 instances should be considered as "re-discovered". Apparently, Android Lollipop tries to fix the security problems caused by the residues in the `Download` service and `Keystore` service, and its inline comments led us to reproduce the attacks on Android KitKat and prior versions. Such discoveries would not be possible without analyzing the code. Those patched vulnerabilities, on one hand, imply Google's awareness of particular data residue instances. On the other hand, they demonstrate the challenges involved in automating the detection process, as Google fails to address all instances. Due to the lack of a full understanding of the data residue problem, Google even repeated the data residue vulnerability in the newly introduced system service called `TrustAgent`.

Based on the intention of the data, we group all residue instances into five categories: *Credential Residue*, *Capability Residue*, *Settings Residue*, *Permissions Residue*, and *History Residue*. For each category, we examine its accessibility, modifiability and utilizability. The examination process starts with the residue detection, followed by hypotheses, and eventually leads to individual experiment design. Since most of the data residue instances identified were previously unknown, there is no existing attack. Therefore, we designed experiments to demonstrate the feasibility of exploits and show the potential damage. To make attacks more realistic, as an important principle in our experiment design, we avoid declaring suspicious permissions in the attack apps. Actually, apps with desired capability already exist in various app stores, as shown in Section V, although they are not attempting any attacks simply due to the lack of knowledge on the vulnerabilities discussed in this paper.

In the following subsections, we will explain the technical details of each attack and our experiment results. For the successful attemtps, we further discuss their preconditions and feasibility in real-world scenarios. Besides that, failed attacks are also important pieces in our research process, as they show how we systematically evaluate the potential damage for each data residue instance. Despite the negative results, all failed experiments are based on valid assumptions, and the insights we learned from them are valuable.

A. Credential Stealing

The popularity of client-server apps on mobile platforms brings in necessity in supporting secure authentication and communication at the framework level. In response, Android uses a system service called `AccountManager` to manage user's online account credentials; it uses another system service called `Keystore` to store the public/private Keypairs for secure communication. Both services store the user credentials on behalf of apps. Although Android carefully restricts the access to these sensitive credentials, both system services are vulnerable to the data residue attack.

1) AccountManager: There are normally two ways for Android apps to authenticate users' online accounts. The first approach requires the client app to provide its own login activity for users to type username and password. This is a concern if the client app and the server do not belong to the same party. Android provides an alternative approach using the `AccountManager` framework, so the client app can be authenticated to the server without knowing the user's credentials.

In this framework, the actual authentication is handled by authenticators, which are installed on the device as trusted apps. Each authenticator defines the account type it can support. For example, in Figure 2, App A is an authenticator app that declares the account type "XYZ". The client app sends requests to `AccountManager` with the account type it wants to authenticate with. The account type allows `AccountManager` to select the corresponding authenticator. In response, `AccountManager` presents a consent UI to the user with information of the requesting app and the authenticator. After user approval, if the corresponding account has not been set up yet, `AccountManager` invokes the login activity within the authenticator app. The user enters username and password once per account into the authenticator, which conducts the actual authentication logic with the remote server. Upon a successful authentication, the authenticator usually returns an OAuth token to `AccountManager`, which further forwards the token to the requesting app.

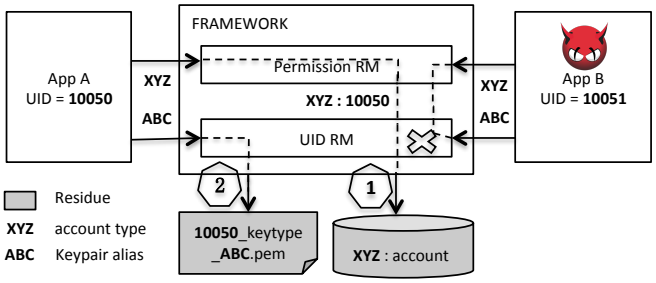

Fig. 2: Android's Protection on Accounts and Keypairs

To avoid asking the user to type his/her credentials repeatedly, authenticators often save the user credentials in `AccountManager`. For future authentication requests, authenticators directly retrieve the saved user credentials from `AccountManager` without launching the login activity again. The authenticator that saves the credentials for a particular account is considered as that account's owner. `AccountManager` only gives the credentials to their rightful account owner, not to others. In Figure 2, although App B declares the required permissions and even the same account type as App A, its UID does not match with the account owner's UID record in `AccountManager`, so if B tries to get the credentials of the account "XYZ", `AccountManager` will deny it.

A-1. Individual Authenticator - No Residue

Because sensitive user credentials are saved by `AccountManager` on behalf of the authenticator, it is important to know *how the authenticator saves account credentials and whether they will be cleaned up after the authenticator is uninstalled*. We designed our experiment targeting a popular app called `myMail`, which has millions of downloads from GooglePlay. This app provides authenticators for a number of accounts, such as Microsoft Exchange and Yahoo. We have observed that passwords for these accounts are saved in plaintext inside `AccountManager`. Many other apps, such as `MeetMe` (with 10 million downloads), have a similar behavior. This is not a concern since `AccountManager` is trusted and the credentials are protected. Moreover, when the authenticator app is uninstalled, the credential data are cleaned up. `AccountManager` does so by checking whether the account type still has a valid owner, and if not, the related data will be deleted. Therefore, it seems that there is no residue problem.

A-2. Duplicated Account Type - Successful Attack

`AccountManager` only deletes the credential residue if its associated account type does not have a valid owner. The interesting question is *whether two unrelated apps could declare the same account type, and if so, whether that can prevent AccountManager from removing user credentials*.

Experiment Design We still targeted the `myMail` app, which declares an account type called `com.my.mail`. We wrote a malicious authenticator app, which declares the same account type. We installed `myMail` first and then our malicious app. Interestingly, at this stage, only the first installed authenticator (`myMail`) is considered as the owner of that account type, and will be in charge of future requests to that account type. Naturally, it can directly access that account's credentials; the same access from our malicious authenticator app will be denied by `AccountManager`'s protection mechanism.

We then uninstalled `myMail`. Surprisingly, Android makes our malicious app the owner of the account type, enabling it to retrieve the user credentials for all the email accounts set up in `myMail`, essentially letting our app inherit `myMail`'s credential residue. This security breach is in `AccountManager`'s cleanup logic, which checks whether the account type to be cleaned up is declared by anyone else; if one is found, `AccountManager` makes it the new owner of the account type. The underlying assumption is that, those who declare the same account type should belong to the same party (e.g. apps with the same signature). Unfortunately, this assumption is not guaranteed.

It should be noted that even if `myMail` only saves the hash value of user credentials, it does not help much; because the attacker can simply copy the information into the `AccountManager`'s database in his/her own rooted device. As long as the app server does not associate hash value with the device, attacker can still get control over the entire account. Actually, `myMail` saves the hash of user's Gmail account password, but we were still able to login to that Gmail account by replicating that hash value onto a different device.

Discussion In order for the above attack to succeed, the malicious authenticator needs to be installed after the target one. This constraint is greatly relaxed, as each authenticator can declare multiple account types, empowering one malicious app to target multiple authenticators using one codebase. Once the precondition is met, the malicious authenticator can behave normally until the target one is uninstalled. Actually, we have observed significant improvements in `AccountManager`'s security specification in the upcoming Android Marshmallow [1].

2) Keystore: Android `Keystore` provides and stores strong cryptographic keys to/for apps during the runtime; it keeps tracks of the keys' ownership using the app's UID, so an app cannot get other apps' keys. In Figure 2, a Keypair named "ABC" is created for app A with UID 10050, thus App B cannot access the pem file because of UID mismatch. Unfortunately, Android fails to clean up the Keypair after an app is uninstalled. As a result, we suspect that, *if a newly installed app has the same UID as the one uninstalled, it may be able to get the Keypair*.

Experiment Design Android Lollipop does clean up the Keypair residue correctly, but its inline comments lead us to believe that the cleanup was incorrectly implemented in prior versions. To confirm that, we switched to KitKat. We first installed `Microsoft Remote Desktop` app on the device, which has Microsoft Azure Active Directory Authentication Library (*ADAL*) embedded [2]. ADAL provides the support for Work Accounts to third-party Android apps. Internally, the app relies on Android `Keystore` to save app specific self-signed certificates and uses asymmetric cryptography to protect the session key for encryption and keyed hash. The Keypair generation is triggered when users sign in to Microsoft Azure. We then uninstalled `Microsoft Remote Desktop`. It turns out that KitKat does not delete the Keypair. After rebooting

the device, we installed our malicious app and were able to get the same UID as the one uninstalled. As a result, our app is able to steal the Keypair left by `Microsoft Remote Desktop`. Similarly, if our malicious app is installed first and then uninstalled, followed by the installation of `Microsoft Remote Desktop`, ADAL will always use the Keypair that the malicious app intentionally left inside `Keystore`.

Discussion The attack above requires the malicious app and its target one to share the same UID after device reboots. However, since the Keypair residue will be kept on the device unless user resets the phone, the incubation period can be quite long. Moreover, Android does not requrie any permissions from the apps to use the `Keystore` feature, allowing apps to easily hide their malicious intention.

B. Capability Intruding

To provide richer user experiences, it is necessary for apps to share resources and functionalities. However, the UID-based access control makes such sharing difficult, because an app's privilege is decided by its UID, which does not change. Capability-based access control is a better choice for achieving sharing. A capability is a token/ticket, which allows its holder to conduct an operation on a particular object, regardless of who the holder is, as long as it is a rightful holder. Because a capability does not bind to any specific subject, it can be delegated to another app, and therefore achieves the sharing purpose. File descriptors are examples of capabilities enabling the holder to conduct operations on files. File descriptors can be passed from a process to its child process, or from one process to another unrelated process using the Unix Domain Socket.

Building upon the capability mechanisms provided by the underlying Linux kernel, Android introduces two types of capabilities at the framework level, one for data sharing and the other for functionality sharing. (1) The most common way to share data on Android is via *content provider*. Content providers manage the access to a structured set of data, and they are the standard interface that connects data in one process with code running in another process. Underneath the implementation, the sharing is achieved using file descriptors, passed to another process via the Unix Domain Socket. However, at the framework level, Android abstracts out the low-level details and presents content provider data to external apps with URI references. Each URI reference consists of two parts: *authority* and *path*. *Authority* uniquely identifies the content provider on the device, and *path* points to a specific table inside the database. Therefore, URI reference serves as the framework-level capability. (2) Functionality sharing enables one app to interact with another app, allowing the first app to leverage the functionality of the second one. Android uses binder token as the capability to enable such interactions. Direct use of binder token is allowed but not easy inside apps, so Android provides a framework-level abstraction called *Intent*, which is built on top of binder and more convenient to use.

Just having capabilities is not sufficient for sharing; Android needs a convenient way to delegate them. Instead of using the low-level Unix Domain Socket mechanism, Android implements three high-level delegation channels. (1) Intent is the most common carrier for capability delegation, and it encapsulates the capability inside its payload section. (2) Binder token itself can also be used for delegation. (3) Another way is Android Clipboard, which allows an app to share the URI capability with multiple recipients.

The extensive usage of capability delegation in Android can potentially lead to data residues, i.e., the capability held by the recipients may remain inside the system even if its owning app has been uninstalled. Handling these capability residues correctly is extremely important; if not carefully handled, these seemingly "dead" capabilities may be "revived" by malicious apps, and used to escalate their privileges. We systematically examined six combinations of two capability types (URI and binder token) and three delegation channels (Intent, binder token and Clipboard). Two combinations are invalid: Android does not support putting binder token on Clipboard, and sending URI reference via binder token does not actually delegate the capability. Among the four valid combinations, one of them is subject to the data residue attack. Even for the failed ones, we would like to answer why they failed, because such information is beneficial to future development.

B-1. URI on Clipboard - Successful Attack

Android provides a clipboard-based framework called Clipboard for copying and pasting. It supports both simple and complex data types. During the copying, simple text data are copied directly to Clipboard; complex data must be stored in a content provider, and its URI reference is copied to Clipboard. Basically, by placing a URI reference on Clipboard, an app can share its data with other apps. An interesting question is what will happen to that URI reference after the app that owns the data is uninstalled. As mentioned before, a capability contains an object ID that identifies the resource associated with the capability. In the URI case, the object ID is *authority*, which is the ID for identifying content providers. Android ensures that an app can only place a URI on Clipboard if it can access the content provider. After the owner of the content provider is uninstalled, obviously, the content provider is deleted as well, so the URI capability refers to a content provider ID that does not exist anymore. Our hypothesis is that *if a newly installed app uses the same provider ID as the one that has just been uninstalled, the URI residue on Clipboard may be used to access the content provider in this new app.* If this hypothesis is true, it can be used to attack newly installed apps.

It should be noted that Android does not allow two apps to declare the same content provider ID on the same device, so the ID is unique. However, if the one who declares an ID is uninstalled, the newly installed app can declare that ID. Thus, the uniqueness is maintained at any point of time, but not throughout a duration. This fact will be the basis for our attack experiment.

Experiment Design In our experiment design, we target the email content provider inside `Yahoo Mail` app, which has more than 100 million installs from Google-Play. The app sets its email provider as private, but with `grantUriPermissions` flag set to true. This means other apps cannot directly access the email provider, but `Yahoo Mail` can create a URI capability, and pass it to the authorized apps, allowing them to access the emails inside the provider. Our objective is to forge a capability, so we can access the emails in `Yahoo Mail`, without being authorized.

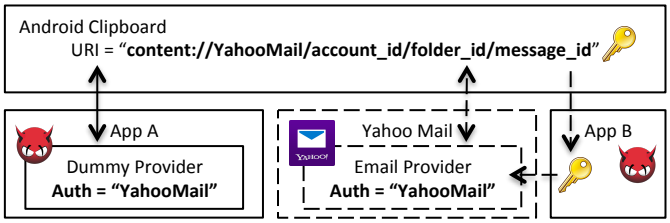

Fig. 3: Yahoo Mailbox Intruding

Our experiment involves two malicious apps, App A and its companion App B. App A needs to be installed before the `Yahoo Mail` app is installed. In App A, we create a content provider that has the same authority (i.e., ID) as the one used in the `Yahoo Mail` app. App A then constructs a URL capability for this content provider, and places the capability on the Clipboard. At this moment, whoever retrieves the capability from the Clipboard can access App A's content provider. This step is depicted in Figure 3 using solid lines with a sample URI value on the Clipboard. Now, App A's job is to keep annoying the user, so eventually it is uninstalled by the user. However, the capability residue is still on the Clipboard.

We then installed the `Yahoo Mail` app. After the installation, inside the companion app B, we retrieve the URI from the Clipboard and resolve it. Interestingly, we are able to successfully access the emails inside the `Yahoo Mail` app, as shown in Figure 3 using dash lines. This is because the ID for `Yahoo Mail`'s email provider is exactly the same as the one used in App A, and Clipboard mistakenly associates the capability residue with the newly installed content provider. This is a security breach; essentially, a capability can be forged with the help of Clipboard.

Discussion There are two preconditions for this attack to succeed. First of all, a malicious app and its companion app have to be installed on the device before the target one. Although this requirement seems to be relatively strong, it still has the chance to be met in practice, as Android Clipboard is publicly accessible with no permission requirements. Moreover, the malicious app can also declare multiple authorities to increase the target scope. The second precondition is that, the target app must be installed on the device after the malicious one is uninstalled. Its likelihood depends on the lifespan of the residue data on Android Clipboard. As long as the installation of the target app happens before another copy operation is performed or the device is rebooted, this precondition can be true. Despite all exploit efforts involved, the existence of such capability residue endangers users' privacy.

B-2. URI in Intent - Failed Attack

URI reference can also be passed to another app using Intent. Therefore, it is intriguing to see whether the above attack works for this delegation channel. We repeated the previous experiment on the `Yahoo Mail` app, but this time, the capability residue is the URI reference sent from App A to App B. Interestingly, the attack failed. A further investigation reveals the subtle but significant difference between these two delegation channels. When a URI capability is sent using Intent, the capability will be bound to the UID of the sender. Namely, the object ID on the capability consists of a tuple: UID and content provider ID. In the Clipboard case, capability is not bound to UID, so it only consists of the content provider ID, making re-association to different UIDs possible. Therefore, to succeed in the attack using the Intent channel, the `Yahoo Mail` app has to be assigned the same UID as App A. As we mentioned before, this is possible, but it requires a system reboot. Naturally, the URI capability, which only exists in memory, will be naturally cleaned up when system reboots.

We further find out that Android supports persistent URI capability, which is saved on disk, and can thus persist after rebooting. Android does a good job cleaning up this form of capability when its owner is uninstalled.

B-3. Binder Token in Intent - Failed Attack

Apps usually do not pass binder tokens directly via Intent, unless the token is a `PendingIntent`. By giving a `PendingIntent` to another app, the grantee allows the receiver app to perform the specified operation using the grantee's permissions and identity. Basically, `PendingIntent` serves as a capability for delegating privileges. `PendingIntent` is quite useful in Android's notification framework: apps need to provide a `PendingIntent` when sending a notification to the system; upon user's click on the notification, Android fires an intent using the app's identity (not its own), avoiding potential privilege escalation.

After an app sends a `PendingIntent` to another app, and it gets uninstalled, the `PendingIntent` will become capability residue. It is interesting to see whether the residue can be used for attacking newly installed apps, like what we did in the Clipboard case. It turns out that although the residue is still left in the system, Android disables the capability when its owner is uninstalled. Therefore, the attack fails.

B-4. Nested Binder Tokens - Failed Attack

In Android, apps can also pass a binder token (a form of capability) directly through the existing binder channel. Our investigation question is *whether the binder token remains effective even if the creator has been uninstalled*. We designed an experiment with App A binding to App B's service, and thus establishing a binder transaction channel. After that, another binder token created by App A is passed through the channel. However, as we find out, as soon as App A is uninstalled, the binder token becomes invalid. Android does a good job in cleaning up all the binder tokens that are delegated by the uninstalled app.

C. Settings Impersonating

As an open platform, Android offers a variety of extensible frameworks for third-party apps to provide system-level functionalities. An example is the Spelling Checker Framework, which can collect user keystrokes and then rely on a thrid-party app to provide spelling suggestions. As shown in Figure 4, App ABC provides the spell checking functionality using the internal service "xyz".

Since multiple apps providing the same functionality can coexist on the device, the user must explicitly choose one (i.e., setting the preferences) through the Settings app. Preferences are saved in a persistent storage in the form of name-value pair. In Figure 4, when the user chooses App ABC as the

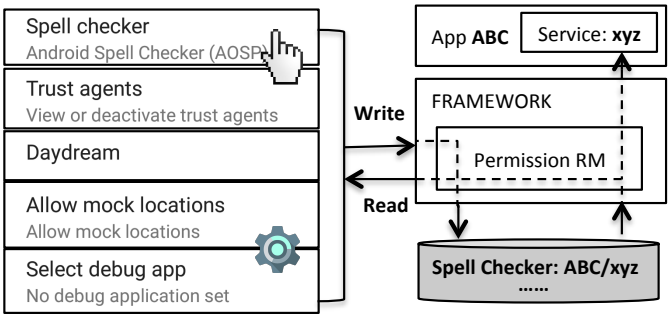

Fig. 4: Android's Protection on Settings Configurations

spell checker, the preference is saved as a combination of the functionality name "Spell Checker" and the service's component value "ABC/xyz".

Android prevents third-party apps from directly accessing security-critical settings. The protection is based on signature-level permissions, and is performed by the permission Reference Monitor (RM) in the framework, as shown in Figure 4. This way, the integrity of the settings is preserved. During the runtime, the system retrieves the preference from the storage, looks up the selected app component, and then authorizes it for privileged operations. A natural question is that, after the selected app is uninstalled, whether its corresponding setting will be deleted, and if not, whether these settings residue can be used for malicious purposes.

We systematically studied all system services that save settings, and found five data residue instances. Due to the page limitation, we only discuss two representative cases to show how the attack works.

C-1. TextService

Android `TextService` is responsible for managing spell checkers on the device; it delivers text inputs to the selected app for spell suggestions. The user needs to select an app as the system's default spell checker, and the selection, which includes a package name and a service name, is saved as an entry in `settings.db`. `TextService` uses this entry to find the selected spell-checker app during the runtime.

After the selected spell-checker app is uninstalled, however, Android does not delete the saved entry from `settings.db`, so the entry becomes a data residue. Our hypothesis is that, *a newly installed app with the same package name and service name can be automatically selected as the default spell checker, without user's approval*. Our experiment confirms this hypothesis. Namely, if the user uninstalls the default spell-checker app, a newly installed app with the same package name and service name will be given all the keystrokes typed by the user, including passwords, credit card numbers, etc.

C-2. TrustAgent

The `TrustAgent` system service, introduced in Android 5.0.0 (Lollipop), provides support for automatic screen unlocking when the environment is trusted. `TrustAgent` relies on an app, called trust agent, to decide whether the environment of the device is trusted or not. For example, users can choose the work place as a trusted environment, so once the trust agent detects that the device is in the work place, it notifies the `TrustAgent` service, which asks the system to relax the security restriction on the device, such as temporarily bypassing the lockscreen.

Users need to explicitly enable a trust agent using the Settings app. This user preference, consisting of the trust agent's package name and service name, is saved in `LockSetting.db` maintained by `LockSettingService`. When the selected trust-agent app is uninstalled, it becomes unclear whether `TrustAgent` or `LockSettingService` should take the responsibility to remove the corresponding entry from `LockSetting.db`. It turns out, nobody takes the responsibility, and the entry becomes a setting residue. Our experiment shows that after the uninstallation of the selected trust agent, any newly installed app can automatically become the trust agent if it has an identical package name and service name as the one uninstalled.

At the current stage, only the app with system signature can be used as a trust agent. Therefore, the above data residue attack does no harm, because the "attacking" app needs to be a system app, which is considered trustworthy. In the future, if Android decides to relax the system-signature restriction on trust agent, this setting residue problem, if not resolved, can lead to damages.

C-3. Other Instances

Several other settings residue instances were identified in our study as well, including debug app, mock location and device dream. The exploiting experiments follow similar patterns to `TextService` and `TrustAgent`. Since the debug app and mock location features are mainly for app testing purpose, we leave out the exploiting details from the paper. In contrast, `dream` is a screen saver launched when a device is being charged and is idle. Different from Desktop screen savers, Android allows the dream screen to be interactive. Thus, the dream setting residue becomes a perfect candidate for conducting phishing attacks. We designed an attacking experiment to exploit the `DreamService` residue through targeting the Airbnb app (10 million installs on GooglePlay). In its dream screen, the `Airbnb` app shows different attractions. We designed a malicious app, namely `Nightmare`, with the same dream component name, but faked `Airbnb` login screen as the dream screen. With the same attack flow, `Nightmare` is automatically enabled as the dream provider, and is thus capable of stealing user's `Airbnb` account credentials through phishing techniques.

Discussion All attack instances mentioned above requires a malicious app to be installed after the target one is removed. This is very likely to happen in practice for two reasons. Firstly, any apps can claim to provide the aforementioned functionality without permission restrictions. Secondly, the residue data will persist in the database and never expires.

D. History Peeking

Android provides system support for commonly used features, such as printing and downloading. For example, an app can send a document to the system for printing. The Print framework considers each request as a "job", and tracks its status. Such history information is saved mainly for two

reasons. First of all, apps may be interested in checking the status of their requests. Secondly, system has to schedule concurrent requests from multiple apps. With various history records spreading all over the Android framework, it would be interesting to know whether these records will be cleaned up after their owners are uninstalled. Our study uncovered three exploitable history residue instances. While the exploit on print record and download history follow the same pattern as for the `Keystore` in Section IV-A2, the process to steal print content is identical to Settings impersonating attacks in Section IV-C. Due to the page limit, we exclude the discussion on their preconditions.

D-1. Print Record

The Android system starts a printing job upon receiving a request from apps. The lifecycle of each printing job includes the following states: *created, queued, started, blocked, completed, failed*, and *cancelled*. Information about the printing jobs will be saved until they are completed or cancelled. If a printing job is failed or not completed, information about this job will be kept in the system, even after the app is uninstalled. Android protects the access to the printing history using the initiating app's UID, so it's the only app that can access the information. We suspect that, *if a malicious app gets the same UID, it will be able to access the information.*

Experiment Design We designed an experiment to test our hypothesis. At the very beginning, `Adobe PDF Reader` app initiates a printing request to the `Google Cloud Print` app, but we intentionally cut off the network connection, making the printing job fail. After `Adobe PDF Reader` is uninstalled, we reboot the device, and install our malicious app called `MyPrint`. This app will be assigned the same UID as `Adobe PDF Reader`. We have observed that `MyPrint` can successfully get the records of all the failed printing jobs created by `Adobe PDF Reader`. Moreover, `MyPrint` is also capable of cancelling or restarting the failed printing jobs.

D-2. Print Content

In the Android framework, the actual printing task is delegated to third-party printer apps. Such a framework accommodates different requirements from printer vendors, such as Canon, HP or Samsung. The user chooses which printer app should be used for printing a particular document. Once a printing task is started, it is associated with the selected printer app's component name. We suspect that, *if the printer app is uninstalled, a newly installed app with the same printing component name will be able to access the failed printing jobs*.

Experiment Design Our experiment setup is the same as above, except that we uninstall the `Google Cloud Print` app instead. After that, the user installs another app named `CustomPrinter`, which has the same printing component name as `Google Cloud Print`. When the user restarts the failed printing job, the task is actually carried out by `CustomPrinter`, allowing this app to access the content of the document.

D-3. Download History

Android keeps each app's download history in the `Download` content provider. Each entry corresponds to a completed download request, and is mapped to the UID of the app that initiates the download, so an app is only allowed to access its own downloaded files. Apps can specify the location for storing the downloaded files, or a default directory in the system's `Downloads` app will be used. Until Lollipop, Android does not delete those downloaded files when their owner apps are uninstalled. We suspect that, *a newly installed app with the same UID can gain the access to the files downloaded by their previous owner*. Our attack only considers files downloaded to the default location (/data/data/-com.android.providers.downloads/cache/); files downloaded to shared folders are public and already accessible to other apps.

Experiment Design We designed our experiment on Android KitKat to target the `DuckDuckGo` app, which is available on GooglePlay with one million installs. It allows users to search information online and download files. Since the download directory is not specified, all the downloaded files will be stored inside the default location. After uninstalling `DuckDuckGo`, we reboot the device, and install our malicious app, which gets the same UID as the previously uninstalled `DuckDuckGo` app. As it turns out, our malicious app can access the contents of all the files downloaded by the `DuckDuckGo` app.

E. Permissions Regaining

Android normally assigns each app a unique UID during the installation, but there are exceptions: apps declaring the same `sharedUserId` value will share the same UID upon successful certificate checks. In this case, permissions granted to these apps are combined to form a "permission pool", and all apps share the same set of permissions from this pool. If an app is updated to a new version with a different permission set (user approval is needed), the "permission pool" will be updated accordingly to add the newly granted permissions, but the ones only declared by the older version (not in the updated version) are not removed, resulting in permission residues. Moreover, when the app is uninstalled, only the permissions declared in the updated version are removed from the "permission pool", which creates a path for privilege escalation.

Experiment Design In order to verify the potential permission residue attack, we designed a sample app named `ContactViewer`, which declares the `sharedUserId` "uid.share" and requests the `READ_CONTACTS` permission. An updated version comes with the same `sharedUserId` value but without requesting any permissions. As we mentioned above, our experiments show that the app still has the `READ_CONTACTS` permission. We then installed another app named `ContactSearch` with the same `sharedUserId` value and signature as `ContactViewer`. Without requesting any additional permissions, it naturally inherits the `READ_CONTACTS` permission granted to "uid.share". We then uninstalled `ContactViewer`. Android is supposed to remove all the permissions granted to `ContactViewer`, but as it turns out, `ContactSearch` can still access the contacts database, indicating that it still holds the `READ_CONTACTS` permission residue introduced by the first version of the `ContactViewer` app. The permission residue can result in over-privileged apps on the device.

Discussion To take advantage of this privilege escalation channel, two apps with the same `sharedUserId` value

Attack Instances		Account	Clipboard	Download	Dream	Keystore	Permission	Print	Spell Checker
I: Analysis on Real-world Applications									
# Targets		131	92	17	24	63	55	49	16
II: Examination on Essential Attributes									
Attributes		account type	authority	UID	package	UID	sharedUserId	UID/package	package
III: Measurement on Device Customization Influence[†]									
LG Nexus 4	5.1.0	✓	✓	✗	✓	✗	✓	✓	✓
Galaxy Nexus	4.3	✓	✓	✓	✓	✓	✓	N/A[1]	✓
ASUS Nexus 7 (2013)	5.1.1	✓	✓	✗	✓	✗	✓	✓	✓
Samsung Nexus S	4.1.2	✓	✓	✓	N/A[1]	N/A[1]	✓	N/A[1]	✓
LG Nexus 5	5.0.1	✓	✓	✗	✓	✗	✓	✓	✓
Samsung Tab 10.1	4.0.4	✓	✓	✓	N/A[1]	N/A[1]	✓	N/A[1]	N/A[2]
HuaWei Y321	4.1.2	✓	✓	✓	N/A[1]	N/A[1]	✓	N/A[1]	N/A[2]
Moto X (2014)	5.0.0	✓	✓	✗	✓	✗	✓	✓	✓
Samsung Note 8.0	4.4.2	✓	✖	✓	✓	✓	✓	N/A[1]	N/A[2]
LG G3	5.0.0	✓	✓	✗	✓	✗	✓	✓	N/A[2]

[†] N/A[1]: feature Not Available because of the low Android version; N/A[2]: feature Not Available because of the vendor customization.

TABLE II: Impact of Android Data Residue Vulnerability in Practice

should coexist on the device. Actually, it is quite common for developers to submit multiple apps to the appstore. According to [5] in 2011, the average number of apps submitted per developer is 6.6 in the Android Market. With the recent auto-update feature on Android, the exploit likelihood increases.

V. Evaluation

In this section, we evaluate how the data residue attacks may potentially affect the real world. To this end, we plan to evaluate (1) the impact of the attack on real-world apps, (2) the feasibility for the malicious apps to be uploaded to app markets, such as GooglePlay, Amazon Appstore and Samsung Appstore, and (3) how the vendor customization affects the attack. Since the damage of the mock-location residue and the `Debug` setting residue is marginal, we exclude them from our analysis. Our evaluation results are summarized in Table II. In the rest of this section, we report the details of our evaluation.

Analysis on Real-world Apps We perform a large-scale analysis on 2,373 unique apps (top 100 free apps in 27 categories) collected from GooglePlay in March 2015. Our static analysis is built upon the AndroGuard framework [3] and consists of two steps. The first one is to detect apps with the usage of Android system services that are vulnerable to the data residue attacks. This can be done by matching specific permission and component declarations from apps' manifest files or Android APIs from decompiled source code. For instance, the declaration of a service component listening to `SpellCheckerService`-typed intent action with `BIND_TEXT_SERVICE` permission requirement makes this app a spell checker. The second step is to examine whether the triggering conditions can be applied to this particular app. To illustrate, consider the `DownloadManager` service. We flag an app as providing the download functionality if the `DownloadManager.enqueue()` API is found in its codebase. However, in order for it to be exploitable, the app needs to save the downloaded files in the default directory. As a result, we further excluded apps with APIs that can customize the download directory.

The final results in Table II(I) indicate that numerous Android apps can be affected by the data residue vulnerability. As each app comes with millions of downloads, the damage is quite significant. Among these apps, 131 apps act as authenticators and 63 apps use Android `Keystore`, so if they are uninstalled from the device, user's credentials can be stolen by adversaries. Another attack with severe damages is the capability intruding attack via Android Clipboard. This attack requires the target app to contain a content provider with the `grantUriPermissions` flag set to true. In our analysis, 92 apps satisfy this requirement, and can be the victim of the data residue attacks. The data that can be leaked are quite sensitive, including files in the cloud (`OneDrive`, `Box`, `Dropbox Photos`), financial statements (`Chase`, `Walmart`, `Progressive`), social information (`Tango`, `Contacts+`), etc. Moreover, the settings impersonating attack affects 40 apps, including 16 spell checkers and 24 dream providers; the history peeking attack affects 66 apps, including 17 apps due to the download feature and 49 apps due to the printing feature.

Assessment on App-store Defense A closer look at all the data residue instances reveals that, Android's existing protection implicitly depends on the uniqueness of several attributes. We map out the essential attributes for successful attacks in Table II(II). Our experiments have demonstrated the possibility to break the uniqueness on the device. However, it is unclear whether the uniqueness is preserved when apps are uploaded to app markets. Because the defense of app markets can only check the static information in the apk file, attributes that are dynamically determined during the installation (e.g. UID) are beyond its control. Therefore, we focus on these three attributes: account type of authenticator, authority of content provider, and package name of app.

Our assessment results indicate that, none of the appstores perform uniqueness checks on account type or authority. To be more specific, we are able to detect existing authenticators with the same account type, and upload apps to all three appstores with duplicated authority names. At the same time, we have observed that individual appstore will preserve the uniqueness

of package name. However, the target package name may exist only in particular appstores, allowing attackers to upload apps with the same package name to other appstores.

Measurement on Device Customization As we have mentioned before, our study is based on the analysis of the official Android Lollipop codebase, which, however, may be customized extensively by various vendors to fit their needs. To measure how the vendor customization affects the data residue attacks, we repeated 8 attacks on 10 different devices running different versions of Android. The test results are summarized in Table II(III).

Not all features are available on every device. For example, `DreamService` was first introduced in version 4.2.2, while the printer support was recently added in KitKat. Moreover, some vendors remove certain features from their devices. For example, the spell checker feature is removed from most Samsung devices. Because of these reasons, there are only 65 valid attack attempts. Among them, 54 (83%) attacks are successful. For the 11 failed attempts, 10 of them are caused by the fixes introduced in Lollipop (regarding the `Download` and `Keystore` residues). Actually, the exploits on download residue still have the chance to succeed on devices running Android 5.0.0 and above, but only if the device is set up for multiple users. As a result, we do not consider them as successful attempts. The only case not caused by these fixes is the Clipboard exploit on Samsung Note 8. The customization on this device reduces the power of URI permissions, which results in security exceptions during our attacks.

VI. Discussion

Data residue in the Android system is a challenging and unique problem to solve. To see why it is unique, let us compare with the traditional desktop environment. First, the data residue problem is created when a user is deleted from a system. In the traditional environment, deleting users is not very frequent, but in Android, each app acts as a different user, so app uninstallation basically involves deleting an existing user; such user-deletion occurs much more frequently than in the traditional computing environment. Second, in mobile systems, apps work in a much more collaborative manner than those in the traditional systems. Namely, mobile apps depend on other apps to fulfill some of the functionalities, such as spell check and authentication, instead of implementing those functionalities all by themselves. Android provides many system services to facilitate such a collaboration, which inevitably leads to app data (i.e. user data) being stored outside the app's storage space.

The high frequency of "user" deletion and the wide spreading of "user" data across the system make data cleanup a very challenging task during app uninstallation. Therefore, data residue is more likely to occur in Android (and other mobile systems) than in the traditional systems. It is imperative that the design of system services should explicitly address the data residue problem. The design should clearly specify whether there is a potential data residue, whether app's data are removed when their owner apps are uninstalled, and if not, what security consequence might occur if the data are inherited by other apps.

Android has addressed the data residue concern in the design of some of the system services. For instance, the recent Android version, Lollipop, has fixed the residue problem inside the `Download` and `Keystore` services. However, without a systematic study on all residue instances in the system and their fundamental causes, the solutions are ad hoc and only work for individual cases. For example, while the above two instances are fixed, a new data residue problem has been created with the introduction of `TrustAgent` service. A generic solution should be based on a thorough understanding of the problem. Our research made the first attempt towards a systematic understanding of the data residue problem in the Android system. Clearly, significant additional efforts need to be made to solve the problem completely. We hope that the research community can build upon our understanding, and develop effective solutions. In the following part of this section, we present some of our thoughts towards this goal.

Fundamental Causes There are two conditions for data residue to become vulnerabilities: the existence of data residue and finding ways to exploit it. Therefore, if we can remove any of the conditions, the problem is fixed. To avoid leaving data residue requires better software engineering practice, guidelines, development support, and detection tools. This is one direction to pursue in research. Another direction is to identify what can prevent data residue, even if they exist in the system, from being exploited. Android has made reasonable efforts in protecting those data in system services, because it needs to ensure that an app can only access its own data. The protection can be generalized as attribute-based access control, i.e., Android associates each data entry with a corresponding attribute, and then allows the access by the apps that possess the attribute. This access control implicitly assumes that these attributes are unique to individual apps; otherwise, multiple apps can access the same data. Unfortunately, although the attributes in this assumption seem to be unique in the system, there is no guarantee by Android. For example, uninstallation and device reboot can invalidate the assumptions, leading to the re-association of some attributes to a different app. We have already presented the essential attributes used by Android in Table II(II). Here we further summarize their underlying assumptions, protection effectiveness, and breaking conditions in Table III.

Built upon Linux kernel, Android extensively utilizes the UID at the framework level for access control and policy enforcement. The underlying assumption is that two apps cannot possess the same UID at any time. This is true in individual device cycle, i.e., Android does not reuse an app's UID after it is uninstalled, but the assumption does not hold across device cycles. It is possible for a newly installed app to possess a previous app's UID value after device reboots. For package names, Android ensures that apps with the same package name cannot be installed on the same device at the same time, except for the multi-user scenario. However, this does not prevent a newly installed app to use the same package name as the one that was already uninstalled. Android also uses component-based attributes to protect app data. App usually consists of multiple components, which are labeled with component names. For components that provide specific functionalities, such as authentication and structured data storage, Android introduces customized attributes to uniquely identify each of

Layers	Attributes	Assumptions	Protection Effectiveness	Breaking Conditions
Framework	UID	UID exclusion	individual device cycle	device rebooting
Application	package	package exclusion	individual device state	(un)installation
Component	account type	customized-id exclusion	Invalid	(un)installation
	authority	customized-id exclusion	individual device state	(un)installation

TABLE III: Security Examination of Android Attributes Used in Protecting Data Residue

them. Unfortunately, the underlying assumptions are either invalid from the very beginning or only effective at specific conditions.

Defense Based on the above analysis on the fundamental causes of the data residue problem in Android, defense can be implemented in two places: frontend and backend. The frontend protection aims at preventing unauthorized access to the data still left in the system after its owner app is uninstalled. To achieve this goal, the uniqueness of all essential attributes should be preserved across device states and cycles. Such a property requires a record of all the attribute values from the installed apps. When there is an attribute conflict between a newly installed app and an uninstalled app, the user will be presented with an alert and be asked to approve or disapprove. The frontend protection serves as a signature-based preventive system, and its effectiveness relies on the completeness of the signature database. In the data residue case, each signature refers to individual attribute associated with the data. We have explored this idea based on all attributes uncovered in our research, and we are able to defeat all exploits presented in this paper. More importantly, our research urges system architects to carefully select attributes used in restricting the access to sensitive data.

The backend protection aims at eliminating all data residue in system designs. As the first to bring the awareness of the data residue problem to the public, our manual analysis on Android system services may miss sophisticated data residue instances. On one hand, the existing static and dynamic analysis tools on Android mostly stay at the app level and do not fit our needs for examining the framework-level code. On the other hand, some data residue instances can only be triggered under certain orders of conditions, making the automatic detection more challenging. Moreover, all data residue instances uncovered in our research are based on Android AOSP codebase. Although section V shows that vendor customization is mostly based on Android AOSP codebase, and naturally inherits all defects in handling app uninstallation, it remains unclear whether the heavy customization from various vendors at different levels [53], [58] can make the data residue problem worse. Thus, our study intends to provide the baseline of the data residue problem in the Android ecosystem. We expect that a more accurate and comprehensive automatic detection system will be explored by both the academia and the industry in the near future.

VII. Related Work

The popularity of Android has attracted lots of interests from researchers. The main research focus falls on understanding the security landscape of the Android ecosystem, uncovering vulnerabilities in Android apps and system, and enhancing the security architecture of Android. In this section, we review the related prior studies from these three directions and compare them with our work.

Android Security Demystification The Android ecosystem involves several parties, such as developers, apps, Android system, vendors and end users. A high-level view of Android security is presented in [29], followed by Stowaway [32], which maps APIs to their permission requirements. Moreover, the security implication of vendor customization is studied in [15], [53], [58]. Previous studies also examine the third-party ad libraries [35], [40], [45], [49], [56], user involvement [33], [41] and the app installation process [18] on Android. While the knowledge gained from the existing work helps us conduct our study, none of them studied the security risks of the app uninstallation process.

Android Vulnerability Exploration Another line of research is devoted to uncovering vulnerabilities in the Android system and apps. Luo et al. [43] demonstrate attacks on Android's WebView component, while Wang et al. [52] identify unauthorized origin crossing attacks on popular Android apps. The prevalence of content provider vulnerabilities is studied by Zhou et al. [59]. Previous studies [24], [36] also use unguarded public interfaces in vulnerable Android apps to launch attacks. Two recent studies further examine the crypto misuse in Android apps [26], [38]. These vulnerabilities are partially due to developers' mis-configurations of app components or misinterpretation of Android's security protection. The data residue vulnerability identified in our research, however, arises directly from Android system services and demands a framework-level solution.

Prior studies have also revealed several flaws in the Android system. The vulnerability in Android's upgrading process allows a malicious app to escalate its privileges in the new system [54]. Also, the problem of permission revocation at the time of app uninstallation has been discussed in [31], [34], [48], [50]. Exploits on Android Clipboard enable attackers to gain accesses to user's sensitive data [30], [57]. Those vulnerabilities are linked to two specific Android system services, the `PackageManager` service and `Clipboard` service. In contrast, the data residue vulnerability affects a much broader range of system services in Android.

To understand the damage scope of each attack, several static analysis frameworks are proposed for Android, including AndroGuard [3], CHEX [42], FlowDroid [16], Epicc [44], etc. Most of the work is build upon WALA [14] or SOOT [12], but makes extensive customization to model specific system behaviors. These tools mostly stay at the app level and do not fit our needs for examining the framework-level code. In our work, AndroGuard is utilized to identify potential targets on GooglePlay. Dynamic analysis is also widely used in

understanding app's behaviors [39], [46], [47]. Our verification experiments rely on various triggering conditions, such as device reboot, app installation and uninstallation, which are difficult to fully automate using the dynamic analysis approach. Despite all challenges involved, an automatic detection system would be helpful in eliminating all data residue instances from Android eventually, which itself is another research problem.

Android Security Enhancement Several architectures have been proposed to enhance Android security. With SELinux in the kernel as a building block, SEAndroid [51] and FlaskDroid [20] attempt to develop flexible Mandatory Access Control (MAC) frameworks for Android. With the MAC support, a more strict and system-wide policy can be enforced to restrict data accesses. As for the framework level enhancement, TaintDroid [27] applies system-wide dynamic taint tracking and analysis to monitor the flow of sensitive information through Android simultaneously. AppFence [37] is built upon TaintDroid and denies all the unnecessary data request and blocks communications that would lead to privacy leakage. Together with ScanDroid [11], Aurasium [55], XMandDroid [19], DroidChecker [21], PScount [17], WoodPecker [36] and other proposed security frameworks [22], [23], [25], [28] for Android, they strike to either protect user privacy or restrict app's privilege. The common technique in use is statically modeling and dynamically monitoring app's suspicious behaviors. The data residue vulnerability, however, allows newly installed app to possess the data, so the data-access operations appear completely legitimate. The challenge here is to identify all data creation functions and to correctly mark the data with the associated app. Those data could come directly from apps, but also be dynamically constructed within the system services, making the tainting strategy complicated. We leave it as our future work in exploring the possibility of applying MAC policies and framework-level static/dynamic analysis to solve the data residue problem.

VIII. Conclusions & Future Work

In this project, we made the first step towards a better understanding of the security implication in the app uninstallation process, by systematically examining the data cleanup logic within 122 Android system services. Our study uncovered 12 data residue instances, and 11 of them are found to be exploitable in our testing experiments, leading to severe damages. Our work further demonstrates the feasibility of the data residue attacks against real apps, and the attacking apps can be distributed through the existing app markets. To mitigate the threat, clear guidelines should be provided to Android framework developers regarding the data cleanup operation during the app uninstallation process. Further efforts are also needed to design a generic solution for preserving the uniqueness of attributes used by Android framework to save sensitive resources. Actually, Android has already been using a combination of package name and developer key to uniquely associate app data between mobile devices and wear devices. This practice can be generalized to mitigate the data residue risk. Several other approaches can also be applied to defeat the data residue attacks. For example, we can use taint analysis to carefully label and eventually remove all the data residues from the system; we can define mandatory access control policies on data residues to prevent unauthorized access. We will pursue these ideas in our future work.

Acknowledgment

We would like to thank our shepherd, William Enck, and our anonymous reviewers for their insightful comments. This project was supported in part by the NSF grant 1318814. Xiao Zhang and Yousra Aafer thank Samsung Research America for supporting this project during their internships.

References

[1] AccountManager Changelog. https://goo.gl/oD2qXt.

[2] ADAL Android SDK. https://goo.gl/uJGzAU.

[3] AndroGuard. http://code.google.com/p/androguard/.

[4] AndroidCentral Poll. http://goo.gl/nl5z6y.

[5] App Genome Report. https://goo.gl/eGszpB.

[6] AppBrain Statistic on Android Apps. http://goo.gl/CyDYNc.

[7] Demo of the Paper. https://sites.google.com/site/droidnotsecure/.

[8] IDC Report. http://goo.gl/z2AyMV.

[9] iResearch on App Life Expectancy. http://goo.gl/jwENYX.

[10] Localytics on App Retention. http://goo.gl/NWRCGL.

[11] SCanDroid. http://spruce.cs.ucr.edu/SCanDroid/.

[12] Soot. http://sable.github.io/soot/.

[13] Statista Report. http://goo.gl/kkwLW9.

[14] WALA. http://wala.sourceforge.net/.

[15] Y. Aafer, N. Zhang, Z. Zhang, X. Zhang, K. Chen, X. Wang, X. Zhou, W. Du, and M. Grace. Hare hunting in the wild android: A study on the threat of hanging attribute references. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, New York, NY, USA, 2015.

[16] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, New York, NY, USA, 2014.

[17] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: Analyzing the android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, New York, NY, USA, 2012.

[18] D. Barrera, J. Clark, D. McCarney, and P. C. van Oorschot. Understanding and improving app installation security mechanisms through empirical analysis of android. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, New York, NY, USA, 2012.

[19] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technical report, Technische Universitt Darmstadt, 2011.

[20] S. Bugiel, S. Heuser, and A.-R. Sadeghi. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, Washington, D.C., 2013.

[21] P. P. Chan, L. C. Hui, and S. M. Yiu. Droidchecker: Analyzing android applications for capability leak. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, New York, NY, USA, 2012.

[22] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, New York, NY, USA, 2011.

[23] M. Conti, V. T. N. Nguyen, and B. Crispo. Crepe: Context-related policy enforcement for android. In *Proceedings of the 13th International Conference on Information Security*, ISC'10, Berlin, Heidelberg, 2011.

[24] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on android. In *Proceedings of the 13th International Conference on Information Security*, ISC'10, Berlin, Heidelberg, 2011.

[25] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *20th USENIX Security Symposium*, San Francisco, CA, Aug. 2011.

[26] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, New York, NY, USA, 2013.

[27] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, Berkeley, CA, USA, 2010.

[28] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, New York, NY, USA, 2009.

[29] W. Enck, M. Ongtang, and P. McDaniel. Understanding android security. *Security Privacy, IEEE*, 7(1), Jan 2009.

[30] S. Fahl, M. Harbach, M. Oltrogge, T. Muders, and M. Smith. Hey, you, get off of my clipboard. In *In proceeding of 17th International Conference on Financial Cryptography and Data Security*, 2013.

[31] Z. Fang, W. Han, and Y. Li. Permission based android security: Issues and countermeasures. *Computers & Security*, 2014.

[32] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, New York, NY, USA, 2011.

[33] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, New York, NY, USA, 2012.

[34] E. Fragkaki, L. Bauer, L. Jia, and D. Swasey. Modeling and enhancing androids permission system. In *Computer Security ESORICS' 12*, volume 7459 of *Lecture Notes in Computer Science*. 2012.

[35] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, New York, NY, USA, 2012.

[36] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *19th Annual Network and Distributed System Security Symposium*, NDSS' 12, San Diego, California, USA, 2012.

[37] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, New York, NY, USA, 2011.

[38] S. H. Kim, D. Han, and D. H. Lee. Predictability of android openssl's pseudo random number generator. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, New York, NY, USA, 2013.

[39] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, SOAP '14, New York, NY, USA, 2014.

[40] I. Leontiadis, C. Efstratiou, M. Picone, and C. Mascolo. Don't kill my ads!: Balancing privacy in an ad-supported mobile application market. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems and Applications*, HotMobile '12, New York, NY, USA, 2012.

[41] J. Lin, S. Amini, J. I. Hong, N. Sadeh, J. Lindqvist, and J. Zhang. Expectation and purpose: Understanding users' mental models of mobile app privacy through crowdsourcing. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, New York, NY, USA, 2012.

[42] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: Statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, New York, NY, USA, 2012.

[43] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin. Attacks on webview in the android system. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, New York, NY, USA, 2011.

[44] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, Washington, D.C., 2013.

[45] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '12, New York, NY, USA, 2012.

[46] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *21st Annual Network and Distributed System Security Symposium*, NDSS 14', San Diego, California, USA, 2014.

[47] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: Automatic security analysis of smartphone applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, New York, NY, USA, 2013.

[48] J. Sellwood and J. Crampton. Sleeping android: The danger of dormant permissions. In *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, SPSM '13, New York, NY, USA, 2013.

[49] S. Shekhar, M. Dietz, and D. S. Wallach. Adsplit: Separating smartphone advertising from applications. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, Berkeley, CA, USA, 2012.

[50] W. Shin, S. Kwak, S. Kiyomoto, K. Fukushima, and T. Tanaka. A small but non-negligible flaw in the android permission scheme. In *Proceedings of the 2010 IEEE International Symposium on Policies for Distributed Systems and Networks*, POLICY '10, Washington, DC, USA, 2010.

[51] S. Smalley and R. Craig. Security enhanced (SE) android: Bringing flexible MAC to android. In *20th Annual Network and Distributed System Security Symposium*, NDSS 13', San Diego, California, USA, 2013.

[52] R. Wang, L. Xing, X. Wang, and S. Chen. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, New York, NY, USA, 2013.

[53] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. The impact of vendor customizations on android security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, New York, NY, USA, 2013.

[54] L. Xing, X. Pan, R. Wang, K. Yuan, and X. Wang. Upgrading your android, elevating my malware: Privilege escalation through mobile os updating. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, Washington, DC, USA, 2014.

[55] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical policy enforcement for android applications. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, Berkeley, CA, USA, 2012.

[56] X. Zhang, A. Ahlawat, and W. Du. Aframe: Isolating advertisements from mobile applications in android. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, New York, NY, USA, 2013.

[57] X. Zhang and W. Du. Attacks on android clipboard. In S. Dietrich, editor, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 8550 of *Lecture Notes in Computer Science*. 2014.

[58] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang. The peril of fragmentation: Security hazards in android device driver customizations. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, Washington, DC, USA, 2014.

[59] Y. Zhou and X. Jiang. Detecting passive content leaks and pollution in android applications. In *20th Annual Network and Distributed System Security Symposium*, NDSS '13, San Diego, California, USA, 2013.