May 2016

UNCOVERING AND MITIGATING UNSAFE PROGRAM INTEGRATIONS IN ANDROID

Xiao Zhang
Syracuse University

ABSTRACT

Android's design philosophy encourages the integration of resources and functionalities from multiple parties, even with different levels of trust. Such program integrations, on one hand, connect every party in the Android ecosystem tightly on one single device. On the other hand, they can also pose severe security problems, if the security design of the underlying integration schemes is not well thought-out. This dissertation systematically evaluates the security design of three integration schemes on Android, including framework module, framework proxy and 3rd-party code embedding. With the security risks identified in each scheme, it concludes that program integrations on Android are unsafe.
Furthermore, new frameworks have been designed and implemented to detect and mitigate the threats. The evaluation results on the prototypes have demonstrated their effectiveness.

UNCOVERING AND MITIGATING UNSAFE PROGRAM INTEGRATIONS IN ANDROID

by

Xiao, Zhang

B.S., University of Science and Technology of China, 2010

Dissertation
Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in
Computer & Information Science and Engineering

Syracuse University
May 2016

ACKNOWLEDGMENTS

Special thanks go to my advisor, Prof. Wenliang Du, for all the guidance during my PhD study. In the previous six years, he has always been patient with my progress in academic research. From the discussions we had, I have gradually learned how to do critical thinking, and more importantly, how to be constructive when new ideas are proposed. His patience and support helped me overcome many crisis situations and finish this dissertation.

I would also like to thank Prof. Pinyuen Chen, Prof. Steve Chapin, Prof. Heng Yin, Prof. Vir V. Phoha and Prof. Edmund S. Yu for agreeing to be on my thesis committee. I am extremely grateful for their time in reading my dissertation and commenting on my views.

I am also grateful to my wife, Yue Zhang, who has helped me stay sane through these difficult years. No matter what decisions I made, I know she is always on my side. Her support and care helped me overcome setbacks and stay focused on my graduate study. We have been through several ups and downs in the last few years, and we will surely face new challenges together in the future.

None of this would have been possible without the love and support from my family. They are always encouraging me to pursue my own dreams, which has led to this great journey of my PhD study.

I am very fortunate to cooperate with several colleagues, including Kailiang Ying, Yousra Aafer, Amit Ahlawat and Zhenshen Qiu. Their efforts have helped to realize the ideas in this dissertation. I would also like to thank Tongbo Luo, Xing Jin, Haichao Zhang and Paul Ratazzi for discussing the initial ideas with me. Last but not least, I thank all my friends for the good time we had together. Our friendship has been an important piece in my life.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

The Android operating system was released by Google in 2008 to compete with Apple's iOS, which dominated the smartphone market at that time. To quickly build up the user base, Android chose to become a more open system in comparison with iOS's rigid scrutinization process. The open nature of Android has proved to be a positive incentive towards faster market growth, with more than one billion accumulated device activations and 81.5% market share as of 2014, according to the report from IDC [1]. In the same year, GooglePlay, the official Android market, reached 1.3 million applications (apps, in short) and more than 50 billion downloads [2–4]. The current Android ecosystem sits on a stage where all major players, i.e., hardware suppliers, device manufactures, carriers, app developers, advertising networks and appstores, coexist on one single device. All those benefits, however, come with the security concerns over unsupervised cooperation among those parties on a single device.

While Android isolates apps from one another with process boundaries and further restricts their privileges with a permission-based security model, its design philosophy encourages the integration of resources and functionalities from multiple parties, even with different levels of trust. For instance, untrusted 3rd-party apps can request access to users' contact data managed by the trustworthy `Contact` app preloaded on the device. Moreover, apps released by developers with different security awareness can cooperate seamlessly with the Android framework as the bridge. Another example is the advertising component

frequently integrated in free apps. Users tend to have a lower trust of those advertising components, as they are shown to aggressively harvest users' privacy [5, 6]. However, once embedded into host apps, the trust gap between both parties is diminished from the system's perspective.

Such program integrations, on one hand, connect every party in the Android ecosystem tightly on one single device. On the other hand, they can also pose severe security problems, if the security design of integration schemes is not well thought-out. Given the sensitive resources and functionalities available on mobile devices that can be potentially shared, it is necessary to evaluate, and more importantly, improve the security design of all integration schemes on Android.

1.1 Program Integration Schemes on Android

There are normally two approaches for program integration: code wrapping and through middleware. For code wrapping, one program directly embeds the code of another program and repackages them to a new software. For the middleware approach, those two programs are still separated. Their interactions are facilitated with the middleware in between. On Android, these two approaches have led to intra-app integration and inter-app integration, respectively. As illustrated in Figure 1.1, intra-app integration refers to the sharing among components within the same app, while inter-app integration focuses more on the interaction among apps. Android provides several schemes for both types of integrations. For intra-app integration, developers can directly incorporate 3rd-party code, such as social plugins and advertising SDKs, into their apps.

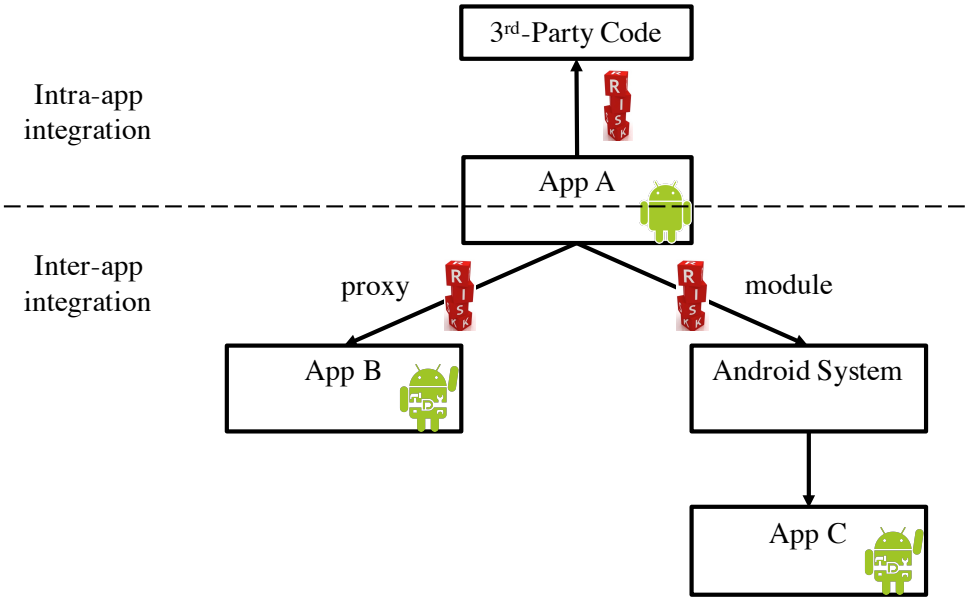


Fig. 1.1.: Program Integration Schemes Available on Android

With the Android framework as the middleware, inter-app integration on Android has provided another two schemes: framework proxy and framework module. For the framework proxy scheme, the Android framework delegates the request originated from app A to the destination app B. Thus, app A can invoke the functionality provided by app B. For the framework module scheme, Android allows 3rd-party apps to be plugged into the system as modules for providing additional security-critical features. An example is Android's Spelling Checker Framework, which can collect user keystrokes and then rely on a 3rd-party app to provide spelling suggestions.

What can go wrong Each integration scheme mentioned above serves as a bridge in connecting two parties with different levels of trust on one single Android device. During program integration, proper access control has to be enforced in order to guarantee security. Otherwise, adversaries can take advantage of flawed integration schemes to

weaken the security of the entire Android platform. This dissertation attempts to answer the question whether program integrations are secure on Android by evaluating the design of those three integration schemes. For that purpose, common risk patterns are first identified for all integration schemes. Then, each integration scheme is evaluated against the identified risk patterns to filter out the ones that have been addressed (details are presented in Chapter 2). Finally, two specific problems are studied in this dissertation:

- For the framework module scheme, after the plugged app is uninstalled from the system, the Android framework still keeps records for that app. The data residue vulnerability arises when the protection on those records becomes ineffective.
- For the 3rd-party code embedding scheme, existing proposals are found to be impractical for separating the privilege of untrusted advertising libraries from host apps.

1.2 Thesis Statement and Contributions

The thesis statement of this dissertation is that, **program integrations on Android are unsafe due to the security risks in integration schemes, including 3rd-party code embedding, framework proxy and framework module**. In support of this statement, this dissertation describes the following contributions:

1. **Data Residue Attacks.** Once an app is selected as a framework module to conduct privileged operations, certain dependencies will be recorded in the Android framework. However, this dissertation has demonstrated that those dependencies

may be left in the system after the app is uninstalled, leading to data residue. Moreover, the protection on those data residue instances will become ineffective. Based on that, this dissertation presents the first systematic analysis of Android's data cleanup mechanism during the app's uninstallation process. The study was carried out in two phases: vulnerability identification and automatic detection. As the first step, the source code of 122 system services on Android 5.0.1 has been manually reviewed with the help of several colleagues. Totally, 12 data residue instances have been discovered. For each identified data residue instance, appropriate hypotheses have been formulated, followed by well-designed experiments to see whether it can be exploited to compromise the system security. A comprehensive evaluation has been conducted to show the impact of each attack on real-world apps, app markets and vendor images.

2. **Data Residue Detection.** As the second step to provide a more accurate and comprehensive understanding of the data residue situation across the entire Android ecosystem, this dissertation presents the design and implementation of ANRED, an automatic ANdroid REsidue Detector. ANRED has been evaluated against 606 Android images from all major vendors and platform versions. The analysis results have confirmed that vendor customization is indeed a major factor in introducing new data residue instances, while the effect of version upgrade varies from vendor to vendor. The effectiveness of ANRED has been demonstrated with 5 new data residue instances identified on the same image that has been manually analyzed in the previous study.

3. **AFrame: Isolating Advertisements from Mobile Applications in Android.** Android permissions are assigned at the app level, so even untrusted 3rd-party libraries, such as advertisement, once incorporated, can share the same privileges as the entire app, leading to over-privileged problems. This dissertation presents the design and implementation of AFrame, a developer friendly method to isolate untrusted 3rd-party code from the host apps. The isolation achieved by AFrame covers not only the process/permission isolation, but also the display and input isolation. The AFrame framework is implemented through a minimal change to the existing Android code base; the evaluation results demonstrate that it is effective in isolating the privileges of untrusted 3rd-party code from apps with reasonable performance overhead.

1.3 Dissertation Roadmap

The rest of this dissertation is organized as follows: Chapter 2 formulates the problem and states the problem scope. Chapter 3 reviews related literature in Android security, isolation techniques and web security. While Chapter 4 presents the data residue vulnerability within Android's app uninstallation process, Chapter 5 describes the design and implementation of ANRED to detect such vulnerabilities on a large scale of Android images. Chapter 6 presents AFrame to mitigate the threat of over-privilege advertisements in Android apps. After that, Chapter 7 summarizes the dissertation. In addition, Appendix A presents another work on Android Clipboard.

2. SECURITY RISKS IN ANDROID PROGRAM INTEGRATIONS

The security of program integrations on Android depends on the design of the underlying integration schemes. Each integration scheme normally involves three components: the initiator, the channel and the target. Although each component may take different forms in different integration schemes, the potential security risk patterns are common. From the perspective of each component, four common risk patterns have been identified as follows.

- **P1**: from the initiator perspective, an adversary can escalate its privilege with the integration of a more privileged target;
- **P2**: from the channel perspective, an adversary can eavesdrop and change the requests passing through the channel;
- **P3**: from the channel perspective, an adversary can exploit the data residue from finished integrations;
- **P4**: from the target perspective, an adversary can steal the sensitive information from the requests;

To answer the question whether program integrations are secure on Android, the design of each integration scheme has been evaluated against all the above-mentioned risk

Integration Scheme	P1	P2	P3	P4
3rd-party code embedding	◐	◐	N/A	◐
framework proxy	○	◐	N/A	○
framework module	○	○	●	○

N/A: risk pattern does not apply to this integration scheme; ○: risk pattern has been resolved by the Android framework or extensively studied in the existing literature; ◐: risk pattern has been studied, but not analyzed or mitigated thoroughly; ●: risk pattern is totally new.

Table 2.1: Security Risks in Android Integration Schemes.

patterns. The results are summarized in Table 2.1, while details are presented in the following sections.

2.1 Framework Module

The framework module on Android can serve as a bridge in connecting multiple apps' functionalities together. To be more specific, Android allows 3rd-party apps to be plugged into the system as modules for providing additional security-critical features. An example is Android's Spelling Checker Framework, which can collect users' keystrokes and then rely on a 3rd-party app to provide spelling suggestions. Another example is the DayDream module, where Android allows a 3rd-party app to show the daydream screen when the device is docked or charging.

To prevent unauthorized apps from using the security-critical features provided by the plugged apps (**P1**), Android commonly restrict the access with permission requirements. Moreover, 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. This also serves to prevent malicious apps from abusing the privileges (**P4**) assigned to the framework module app. Preferences are saved in a persistent storage in the form of

name-value pairs. During the runtime, the system retrieves the preference from the storage, looks up the selected app component, and then authorizes it for privileged operations. The binding between Android framework and the selected framework module app is through Binder token, and thus, prevents adversaries from intercepting the requests (**P2**).

Android also prevents 3rd-party apps from directly accessing security-critical settings. The protection is based on signature-level permissions, and is enforced by the permission Reference Monitor (RM) in the framework. This way, the integrity of the settings is preserved. However, it is unclear whether those preferences are removed after the framework module app is uninstalled from the device. Android has made reasonable efforts to clean up the data owned by an app during the uninstallation process. But 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 (details are presented in Section 4.2.2). Because of the program integration on Android, an app may still depend on the uninstalled one to perform privileged operations (**P4**). Thus, adversaries can take advantage of such dependency to cause damage. In other cases, adversaries can even forge the identity of the uninstalled apps to steal sensitive data from the device.

2.2 Framework Proxy

Android apps can also send requests to or exchange data with other apps using the framework as a proxy. The communication crosses process boundaries twice from the initiator app to the Android framework and the target app. In Android, the Inter-Process

Communication (IPC) relies on the Binder mechanism that features lightweight overhead. One IPC endpoint exposes specific functionality by explicitly publishing a Binder token, which acts as a capability that enables interactions from the other endpoint. Due to the heavy usage of Binder IPC, the Android framework provides a high-level abstraction, Intent, for app developers to use. Intent hides the sophisticated Binder transactions from app developers, and allows them to focus more on providing better combination of functionalities. For instance, instead of implementing the navigation feature, a restaurant recommendation app can invoke the Google Map app for that purpose at runtime.

Android Binder also preserves the identity, i.e., the UID of the initiator is associated with every incoming Binder transaction, allowing the receiver to further enforce security policies. In the example above, the Google Map app can check the identity of the restaurant recommendation app against its policy configuration. However, depending on the security awareness of the app developer, the identity check on the receiver side is not always in place. Thus, adversaries can escalate their privileges by delegating originally unauthorized operations to the vulnerable receiving apps (**P1**). Such attacks are categorized as confused deputy attacks, leading to permission escalation [7], permission re-delegation [8], capability leakage [9] and content leakage [10]. In addition to that, Android itself may decide not to carry, or simply drop the identity information under certain scenarios.

To mitigate the risk above, researchers have proposed solutions from different aspects. AppSealer [11] focuses on vulnerable apps and aims to automatically generate vulnerability-specific patches for preventing component hijacking attacks. In comparison, ComDroid [12], QUIRE [13] and DroidChecker [14] inspect the call chain of IPCs and enforce system-wide security policies. Moreover, previous work [15] attempts to provide a

system-centric IPC provenance on Android, making the caller identity always available on the callee side. From the access control perspective, recent studies, including SEAndroid [16] and FlaskDroid [17], both propose a flexible Mandatory Access Control (MAC) framework for Android. They both enforce mandatory restrictions over all processes, objects and operations based on security labels.

When the Android framework delegates requests to the target app, on one hand, the intended recipient may not exist on the device, resulting in hanging attribute references (Hares). The Hare vulnerability appears when an attribute (e.g., a package/authority/action name) is used on a device but the party defining it has been removed. Harehunter [18] studied the pervasiveness of hanging attribute references (Hares) in popular Android devices. On the other hand, there can be multiple recipients. In some cases, the initiator intends to broadcast the information to multiple parties. In other cases, there may be multiple parties that can provide the functionality requested by the initiator. To avoid malicious recipients from stealing sensitive information from the requests (**P4**), Android explicitly asks the user to make a selection when multiple options are available. The Binder token used in this integration channel will be invalidated when the process of either app dies. Thus, the security risk in **P3** does not apply here.

2.3 3rd-party Code Embedding

Android apps tend to embed 3rd-party code from various sources for different purposes. For instances, one Android app may want to allow users to login with their Facebook accounts. To do that, the app can embed the authentication code provided by Facebook.

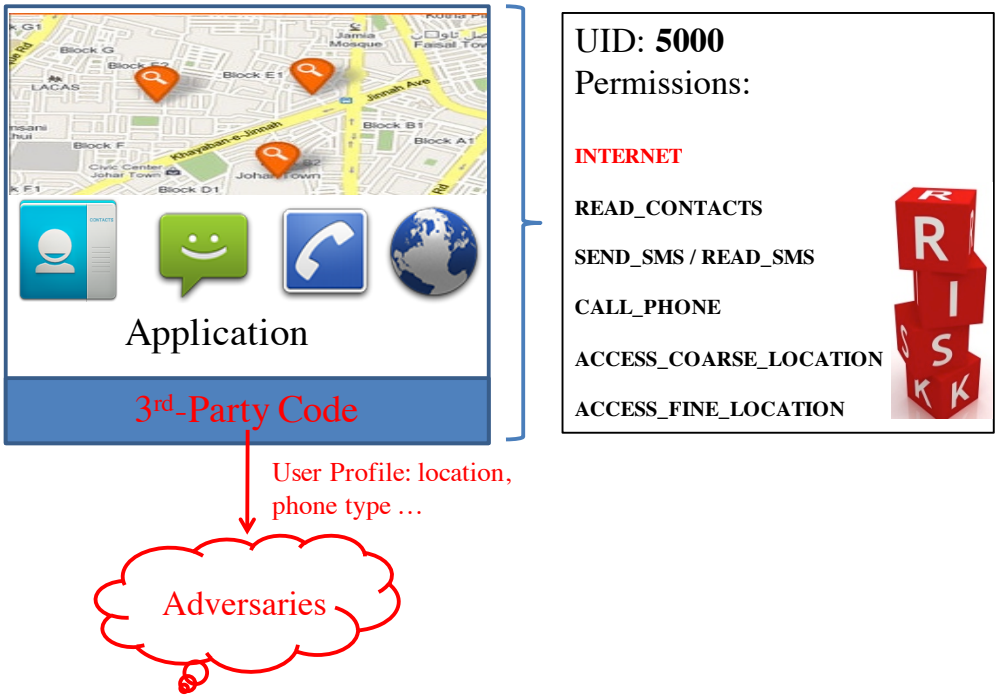


Fig. 2.1.: Security Risks in 3rd-party Code Embedding

Another example is mobile advertising, which is the most common form of intra-app integration on Android. Under the hood, it is a cooperation between mobile advertising networks—such as Google's AdMob [19]—and app developers. Typically, mobile advertising networks distribute a Software Development Kit (SDK) library, and developers can simply incorporate one or several SDKs into their apps. Once incorporated, the SDK code will take care of the communication, advertisement refreshing, and look-and-feel customization. Different from the Facebook case where the 3rd-party code does not affect the user interface of the host app, in-app advertisements commonly have a dedicated region on the screen.

Such program integration is bidirectional, i.e., both the host app and the embedded 3rd-party can be considered as the initiator. Nevertheless, when an app is installed, both the 3rd-party code and the original app will have the same privileges (**P1**), since they are

running in the same process, inseparable by the system. As illustrated in Figure 2.1, the 3rd-party code alone only requires the Internet permission to execute. However, the host app is granted additional privileges like contact access and location access. With the same UID, the 3rd-party code now have the access to users' sensitive data, such as location and phone number, without any restrictions (**P2**, **P4**). In the advertisement case, the advertising code can intercept a user's interactions with the host app, or even inject events to the host app. However, since both sides are running in the same process, their lifecycle events are always synchronized. Thus, the security risk in **P3** does not apply here.

All the security risks mentioned above are because of Android's permission model. In particular, the privileges are granted at the process level, where both the 3rd-party code and the host app are running. Several ideas have been proposed from the research community to mitigate such risks. Apex [20] allows users to selectively grant permissions to apps during the installation. Such restrictions are imposed on the entire app process, and thus, cannot separate the privileges of embedded components from host apps. If the untrusted 3rd-party does not have a user interface, Compac [21] monitors its execution in the Dalvik virtual machine, and enforces access control policies to prevent unauthorized operations. However, the situation is more complicated when 3rd-party code shares device screen with its host app. To mitigate the risk brought by in-app advertisements, AdDroid [22] moves the untrusted 3rd-party advertising code into the Android framework. However, this can only happen if the device vendor reaches an agreement with all advertising companies. Otherwise, apps may fail to show advertisements on certain devices. In addition, AdSplit [23] attempts to isolate the advertising code from its host app using the transparency trick. However, transparency has been widely used by the ClickJacking

Integration Scheme	Risks	Mitigation
Framework Module	spurious connection initiation connection hijacking	Android Binder
	data injection data leakage data modification	Android Binder, system permission, user consent
	data residue (Chapter 4)	**ANRED (Chapter 5)**
Framework Proxy	permission escalation [7] permission re-delegation [8] capability leakage [9] content leakage [10]	IPC provenance [15] IPC inspection [12–14] automatic patch generation [11] MAC policy [16, 17]
	missing target [18]	harehunter [18]
	multiple targets	user consent
	intent sniffing intent manipulation	Android Binder
3rd-party Code Embedding	privilege escalation [5, 6, 25, 26]	Apex [20], Compac [21], AdDroid [22], AdSplit [23], **AFrame (Chapter 6)**

Table 2.2: Project Scope and Contributions.

attack and its variations [24], resulting in a lower possibility for AdSplit to be adopted in practice. Thus, the risk remains in the 3rd-party code embedding scheme.

2.4 Problem Scope

Although there are totally 12 combinations of security risks in all integration schemes on Android, most of them have been addressed by the current Android security architecture or extensively studied in the existing literature. Table 2.2 summaries the identified risks and available mitigation approaches. In this process, the new data residue vulnerability is uncovered. This dissertation further conducts manual analysis (Chapter 4) and static analysis (Chapter 5) to provide a comprehensive understanding of the data residue situation across the entire Android ecosystem. In addition, existing proposals are found to be impractical for separating the privilege of untrusted advertising libraries from

host apps. This dissertation describes the design and implementation of AFrame (Chapter 6) to effectively achieve the designated isolation.

3. RELATED WORK

3.1 Android Security

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

3.1.1 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 [27], followed by Stowaway [28], which maps APIs to their permission requirements. Both works highlight the concern on over-privileged apps and question Android's permission-based security architecture. The AFrame work presented in Chapter 6 also addresses the over-privilege issue, but at a more fine-grained level; at the level of advertising components within host apps. Although the above-mentioned works provide approaches to identify over-privileged apps, they fall short in separating the privilege of embedded components from the host app.

To better understand Android app security, the previous work [25] introduces a `ded` decompiler, which allows security analysts to examine the code logic in Android apps. The

same study also uncovers the pervasiveness of personal identifiers' use in advertising libraries. Motivated by these findings, another line of research tries to reveal more effects of the advertising libraries. For example, the works [5, 6] reveal that in-app advertising libraries actively leak private information to ad servers. Similarly, a recent work [26] shows that important data such as user's interest and demographic information can actually be inferred by ad networks. All the research efforts on understanding the behavior of in-app advertising libraries have motivated the AFrame work in Chapter 6.

Other research works aim to provide a better security understanding of important factors in the Android ecosystem, such as providing a security evaluation of the Android multi-user framework [29], exploring unknown effects of android vendor customization [18, 30–32] and app installation [33], and extracting security-centric descriptions for Android apps [34]. The study on Android data residue presented in this dissertation (Chapter 4 and Chapter 5) is orthogonal to these works with respects to three aspects:

- **Problem**: the data residue work aims to provide a security understanding of app uninstallation process that has never been investigated before.
- **Technique**: the manual analysis of Android data residue (Chapter 4) also relies on source code review to understand the security of the Android framework. In addition, to replicate the data residue attack on real-world apps, their configuration details are also harvested in the evaluation stage.
- **Impact**: the large scale analysis on Android data residue (Chapter 5) contributes to the understanding of vendor customization on Android security.

3.1.2 Vulnerability Exploration

Another line of research is devoted to uncovering and analyzing vulnerabilities in the Android apps. Luo et al. [35] demonstrate attacks on Android's WebView component, while Wang et al. [36] identify unauthorized origin crossing attacks on popular Android apps. Recent work [37] has extended the scope to cover code injection attacks on all HTML5-based mobile apps. These attacks mainly target vulnerable apps that utilize the WebView feature provided by Android. Since in-app advertisement is also hosted in a WebView component, it inherits all the defects mentioned above. The AFrame work presented in Chapter 6 isolates the advertisement into a different process, and thus, limits the damage that an attacker can cause if he exploits a vulnerability in the WebView design.

In addition, the prevalence of content provider vulnerabilities is studied by Zhou et al. [10]. Previous studies [7, 9] also demonstrate the damage of using unguarded public interfaces in vulnerable Android apps to launch various attacks. Two recent studies further examine the crypto misuse in Android apps [38, 39]. 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 this dissertation, however, arises after the app is uninstalled from the device. The problem itself is due to flaws in Android's system services, but endangers the sensitive data stored by the Android framework on behalf of apps.

Prior studies have also revealed several flaws in the Android system. The vulnerability revealed in Android's upgrading process [40] allows a malicious app to escalate its privileges in the new system. In comparison, the data residue vulnerability occurs when an

app is uninstalled, but sensitive data related to this app is still present in the system and acquirable by other apps. Both vulnerabilities allow adversaries to escalate their privileges and gain unauthorized access to users' private data. However, the study on Android data residue in this dissertation not only covers the AOSP codeline, but also evaluates the effects on custom vendor images. Moreover, the problem of permission revocation at the time of app uninstallation has been discussed in [41–44]. The manual analysis of Android data residue (Chapter 4) also uncovers this residue instance. But, it goes beyond that by formulating the data residue problem and conducting a systematic analysis on all system services in the Android framework. Another residue instance identified in Chapter 4 is on Android Clipboard, which greatly benefits from previous studies on Android Clipboard [45, 46].

To understand the damage scope of each attack, several static analysis frameworks have been proposed for Android, including AndroGuard [47], CHEX [48], FlowDroid [49], Epicc [50], etc. Most of the proposed works are built upon WALA [51] or SOOT [52]. To overcome the challenges brought out by the special characteristic of Android framework, they have made extensive customization to model specific system behaviors. In particular, CHEX iteratively identifies and connects broken links in the constructed call graph for Android apps, while Epicc provides a comprehensive set of API pairs in the Android framework that can lead to broken links. FlowDroid focuses on modeling the app's life-cycle events into static analysis engine and looks for privacy leakage in the constructed call graph. In comparison, the design of ANRED (Chapter 5) is based on WALA. ANRED utilizes the scheme from Epicc to identify broken links, but connect them via the bytecode rewriting technique instead of manipulating the call graph.

3.1.3 Enhancement

To improve the security of Android system, researchers have proposed several approaches to enhance Android's framework security architecture, to provide a finer-grained access control model to further restrict apps' privileges and to incorporate mandatory access control into the system design.

Framework Extensions TaintDroid [53] applies system-wide dynamic taint tracking and analysis to monitor the flow of sensitive information through Android simultaneously. AppFence [54] is built upon TaintDroid and denies all the unnecessary data request and blocks communications that would lead to privacy leakage. In this dissertation, manual analysis and static analysis are used to uncover and detect the data residue vulnerability. Additional research efforts are expected to investigate the possibility of employing dynamic analysis to removing data residue in the runtime.

ScanDroid [55] extracts security specifications from apps' manifests, and checks whether data flows through those apps are consistent with the specifications. Kirin [56] certifies each app and verifies the certification at installation time to prevent malware. While ComDroid [12] detects vulnerabilities within apps' communication, QUIRE [13] tracks the call chain of IPCs and enforces system-wide security policies. The data residue vulnerability identified in this dissertation is initiated with the interaction between apps and the Android framework. However, inspecting the IPC channel does not solve the problem as the data leftover has not been generated at the time of communication.

Privilege Restriction. Several works have been proposed to restrict the app's privileges. While Apex [20] allows users to selectively grant permissions to apps during installation, Saint [57] goes further by governing runtime permission use as dictated by an app provider policy. AppSealer [11] aims to automatically generate vulnerability-specific patches for preventing component hijacking attacks in Android apps. All three proposals treat the entire app as one single entity and do not restrict the privilege of embedded components. In comparison, Aurasium [58] applies code-rewriting technique to repackage arbitrary apps with user-level sandboxing and policy enforcement code. However, the protection provided by code-rewriting can be bypassed with native code [59].

Instead, AdDroid [22] and AdSplit [23] provide fine-grained access control at component level. Both of them attempt to restrict the untrusted 3rd-party component (i.e., advertisement) inside the app. To be more specific, AdDroid [22] encapsulates the advertising libraries into the Android framework to lift their trust level. However, this can only happen if the device vendor reaches an agreement with all advertising companies. Otherwise, apps may fail to show advertisements on certain devices. Another proposal is AdSplit [23], which puts the advertisement into a separate activity, with the advertisement activity being put beneath the app activity. However, transparency has been widely used by the ClickJacking attack and its variations [24], resulting in a lower possibility for AdSplit to be adopted in practice. The AFrame work presented in Chapter 6 designs a non-emulated solution without using the transparency technique or changing the original advertising architecture, where advertisements and apps are placed on the same drawing surface, but they run in different processes.

Mandatory Access Control. Recent studies, including SEAndroid [16] and FlaskDroid [17], propose a flexible Mandatory Access Control (MAC) framework for Android. They both enforce a system-wide security policy over all processes, objects and operations based on security labels. On one hand, MAC policies, if configured correctly, can prevent adversaries from accessing the sensitive data residue belonging to a different app. SEAndroid and FlaskDroid can also be utilized to restrict the access to the previously unguarded Android Clipboard (Appendix A). On the other hand, both approaches do not support effectively labelling the advertising components within host apps, and thus, cannot prevent malicious advertising code from abusing the granted privileges. The AFrame work presented in Chapter 6 is able to isolate the advertising component into a different process, thus allowing the adoption of a stricter MAC policy on the advertising process.

3.2 Isolation

Another line of research closely related to the AFrame work presented in Chapter 6 is about isolation. According to [60], existing isolation techniques can be categorized into five groups: language-based, sandbox-based, VM-based, OS kernel-based and hardware-based, as depicted in Figure 3.1. In this section, we review existing isolation techniques available on Android that fall in the above categories. We further present their limitations in achieving the isolation between advertising libraries and host apps. Hardware-based isolation techniques are excluded as only a small portion of Samsung devices have the support for ARM Trustzone.

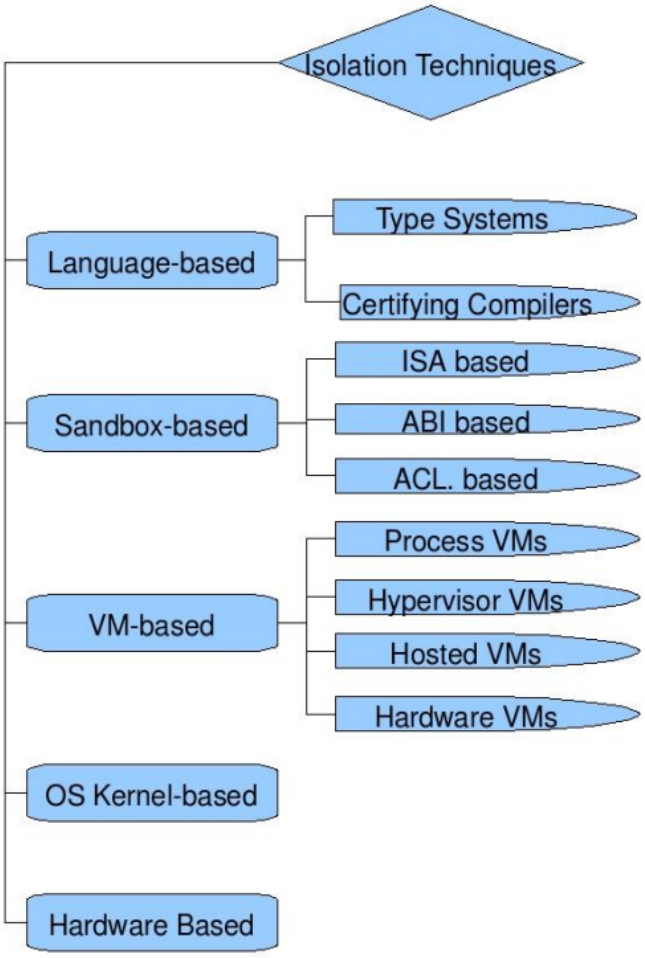


Fig. 3.1.: Taxonomy of isolation techniques [60]

3.2.1 Language-based Isolation

Language-based isolation is an isolation technique provided by programming languages, language compilers, assemblers and/or by runtime environments. There are two approaches that can provide such isolation: type systems and certifying compilers. The first approach (type systems) makes sure that programs can access only appropriate memory locations and that control transfers happen only to appropriate program points [61]. In Android, apps are developed mainly in Java, which is a type-safe programming language. However,

there are two limitations that prevent type system from achieving isolation in Android. First, Android provides NDK [62] as a toolset to allow developers to implement parts of their apps using native-code languages such as C and C++. It is well known that in a conventional setting, native code defeats Java's security, as native code is not covered by Java's security model and has access to the entire address space [63]. Second, previous studies [25, 63, 64] have also demonstrated the presence of native libraries inside Android apps. More importantly, since the embedded libraries and the host apps share the same UID on Android, their privileges are inseparable from the system perspective.

The second technique (certifying compilers) provides a very versatile way of specifying security policies for developers. A widely used example is the javac compiler for Java which produces annotated byte-code from a high-level Java program. Such technique is commonly used in the development of the Android operating system. For instance, `@hide` is widely used in AOSP to prevent apps' access to certain classes/functions at the framework level. Moreover, `@JavascriptInterface` is used to label public methods that are intended to be accessed from JavaScript code. However, the restriction provided by Android annotation can be bypassed with Java reflection. Moreover, it cannot isolate the privilege of 3rd-party libraries from host apps.

3.2.2 Sandbox-based Isolation

According to [60], sandboxing is a technique for creating confined execution environments for running untrusted programs on the same machine. Sandboxes could be implemented based on Application Binary Interface (ABI), Access Control (ACL) and

Instruction Set Architecture (ISA). Android has utilized ABI based and ACL based isolation techniques to restrict the behavior of apps. In particular, the Binder interface (ABI) allows apps to interact with privileged system services or other apps. During the communication, corresponding permissions (ACL) are required from the initiator app in order to access privileged functionalities. However, both ABI and ACL do not achieve a fine-grained isolation at the level of app components. They both treat the entire app as one single entity and do not separate the privilege of embedded code from host apps.

The original Android security architecture does not provide ISA based isolation. However, several research work [65–67] have been proposed to utilize this isolation technique. All proposals are based on bytecode rewriting where additional instructions are added before existing code for restricting usage of security-sensitive APIs [67], improving privacy on Android [66] and enforcing user requirements [65]. However, native code always limits the effectiveness of the proposed solutions. In particular, even though it monitors both direct Java calls and calls from native code to Java methods, AppGuard [65] does not monitor function calls inside of native libraries. Besides native code, previous work [59] has identified a number of other potential attacks targeted at incomplete implementations of bytecode rewriting on Android OS, which can be applied to bypass access control imposed by bytecode rewriter.

The Software Fault Isolation (SFI) [68] is another effective technique aiming at separating distrusted extensions through inserting checking code before every unsafe instruction, like indirect jumps or stores. The fundamental technique is suitable to prevent Android native code within embedded libraries from accessing the memory space of their host app. However, the direct adoption of SFI to Android faces several challenges. First,

3rd-party libraries embedded in Android apps are combinations of Java code and native code, which frequently communicate with each other through the Java Native Interface (JNI) layer. In the advertisement case, advertising libraries can also expose functions to the JavaScript code running inside the WebView component. Thus, it is quite difficult to effectively filter out unsafe instructions from safe ones. Second, Android allows apps to load code at runtime. Existing work [69] has highlighted the risk of unsafe and malicious dynamic code loading in Android apps. From the system perspective, it is challenging to distinguish malicious code loaded by embedded libraries from benign code loaded by host apps. Third, Android has recently replaced the just-in-time (JIT) compiler with an ahead-of-time (AOT) compiler that translates bytecode to native machine code during installation. The native code representation is larger both in permanent storage and in RAM on the device, but the compilation process does not need to be repeated on every application execution. With significant performance improvement, the AOT compiler is also a challenge for adopting SFI, as the runtime context information is missing at the time of compilation.

3.2.3 VM-based Isolation

Virtual machines can also be employed to provide isolation. There are four levels of virtual machines: process VMs, hypervisor VMs, hosted VMs and hareware VMs. At the process level, Android runs each app in its own process, with its own instance of the Dalvik virtual machine. Because of the constrained processing power on mobile devices, Dalvik VM employs the register-based architecture, as opposed to a stack-based Java VM.

However, Dalvik itself is not concerned with runtime security [70]. Existing literature also attempt to bring hypervisor VMs and hosted VMs to Android devices. In particular, Cells [71] introduces a foreground /background virtual phones usage model and proposes a lightweight OS-level virtualization to multiplex phone hardware across multiple virtual phones. Airbag [72] aims to maintain a single phone usage model and the same user experience while providing an isolated runtime environment with its own dedicated namespace and virtualized system resources. However, both proposals target at a restricted running environment for a set of apps. They are not suitable for isolating untrusted components from host apps.

3.2.4 OS Kernel-based Isolation

Android is based on Linux kernel that provides the basic isolation between app processes. For security reasons, Android isolates app processes from one another and from the system by assigning them a distinct Linux User ID (UID) during the installation process. Though effective, such isolation is way too restrictive and prevents legitimate interaction demand between different identities. To bridge the gap and provide fine-grained security features, Android further introduces the permission mechanism. Apps can request access to sensitive resources and privileged operations by declaring corresponding permissions. Upon successful signature check and/or user approval, Android initializes each app's privilege set. Framework level resources are granted via filling in the UID-permission association for this app, while certain hardware related resources, like Internet, Bluetooth and SDCard, are guarded by validating app's Group ID (GID). Granted Permissions

permit apps to gain capability in conducting out-of-sandbox communication. However, permissions are granted at the process level, which can lead to over-privileged 3rd-party libraries embedded in host apps.

3.3 Iframe Security

The AFrame idea presented in Chapter 6 is inspired by the iframe element available for web development. Iframe is an inline frame that places another HTML document in a frame. Browsers keep the context of the iframe and its parent document totally separate by default. Neither the parent document nor the iframe document has access to each other's DOM, CSS styles, or JavaScript functions if they are not from the same domain. Thanks to the provided isolation, iframe is widely used to host advertisements in web pages. The following code shows an example of how to use iframe to embed an advertisement in web pages:

```
<iframe sandbox="allow-popups" src="AD_URL" width="AD_WIDTH"
    height="AD_HEIGHT"></iframe>
```

The isolation achieved by iframe depends on the ability of browsers to separate the origin of Javascript code loaded from different web pages. An origin is defined as a combination of URI scheme, hostname, and port number. The SOP policy prevents a malicious script on one page from obtaining access to sensitive data on another web page through that page's DOM [73]. The origin serves as the identity of script code and allows browsers to enforce additional restrictions. For instance, HTML5 introduces the sandbox

attribute [74] that enables an extra set of restrictions for the content in the iframe. The sandbox attribute allows developers to specify a space-separated list of permissions, including allow-same-origin, allow-top-navigation, allow-forms, allow-scripts, allow-popups and allow-pointer-lock. For instance, the code snippet shown above only allows the embedded advertising code to display a popup window.

The AFrame work attempts to bring the same security guarantee to in-app advertisement on Android. The sandbox attribute available in iframe naturally relates to the permission attribute for Android apps. However, there are two fundamental differences between Android and browser that make direct adoption of iframe challenging. First, the Android security architecture cannot separate the execution of embedded libraries from host apps. In other words, once incorporated, the advertising component runs with the same identity as its host app. To assign a different identity for the advertising component, AFrame isolates it to a separate process with a different UID. However, this leads to the second challenge. At runtime, Android permits only one app (process) to occupy the screen display. With advertising component running in a different process, the screen sharing problem becomes a challenging task in AFrame's design. Actually, we are not the first one facing this challenge. The Chromium Projects [75] also tries to bring support for out-of-process iframes (OOPIFs). As stated in [76], OOPIFs is a massive refactoring effort motivated by the Site Isolation project [77], which allows the browser process to restrict which sites are loaded in each renderer process for security. Despite significant engineering effort, OOPIFs does provide additional security benefit. For instance, native code can be contained inside a different renderer process, as opposed to the traditional NaCl approach [78]. Similar benefit is also available in AFrame's design.

4. DATA RESIDUE ATTACKS ON ANDROID

Android's open design philosophy is fully reflected in the framework module integration scheme, where a 3rd-party app can be plugged into the framework and conduct security-critical operations in a restrictive manner. Such a design pattern is referred as extensible framework and widely adopted in Android system. A representative example is the Spell Checker Framework, i.e., Android allows an authorized 3rd-party app to collect users' keystroke and provide spelling suggestions. Other examples include AccountManager, Dream, Accessibility, Input Method Editor (IME), Notification, Device Administration, Printing, Voice Input and Trust Agent. Despite the different functionalities provided, all extensible frameworks available on Android follow a similar design. The high-level architecture is depicted in Figure 4.1.

As Figure 4.1 shows, Android normally provides a system service for managing the framework module integration between the initiator app and the target app. In this process, the request from the initiator app will be intercepted by the system service and further delegated to the authorized module app. Another common point among Android extensible frameworks is that, the corresponding functionalities provided in the module apps are normally exposed via a service component. To reliably recognize the module app, the above-mentioned system service saves the identifier of the authorized service component and retrieves the information at runtime. Figure 4.1 has further mapped out the common risk patterns presented in Chapter 2. We break down the details about how

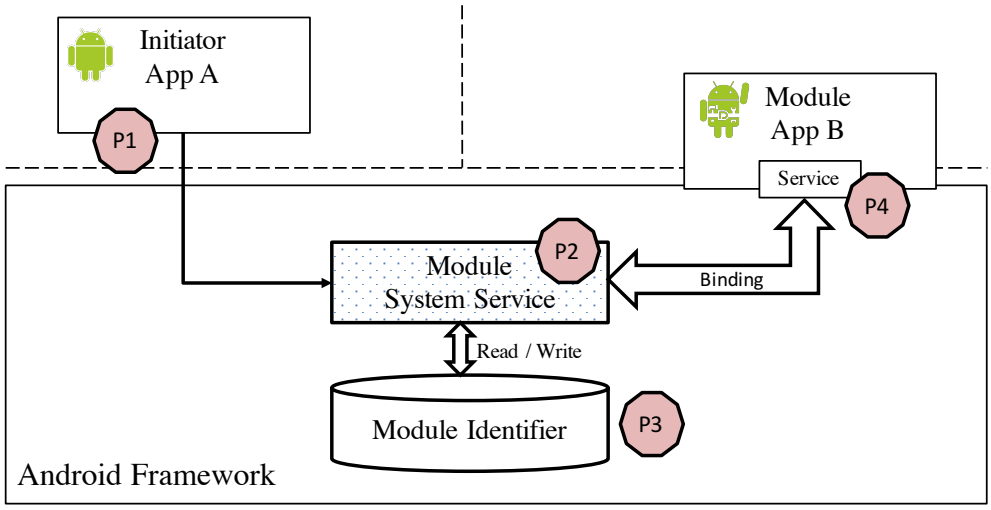


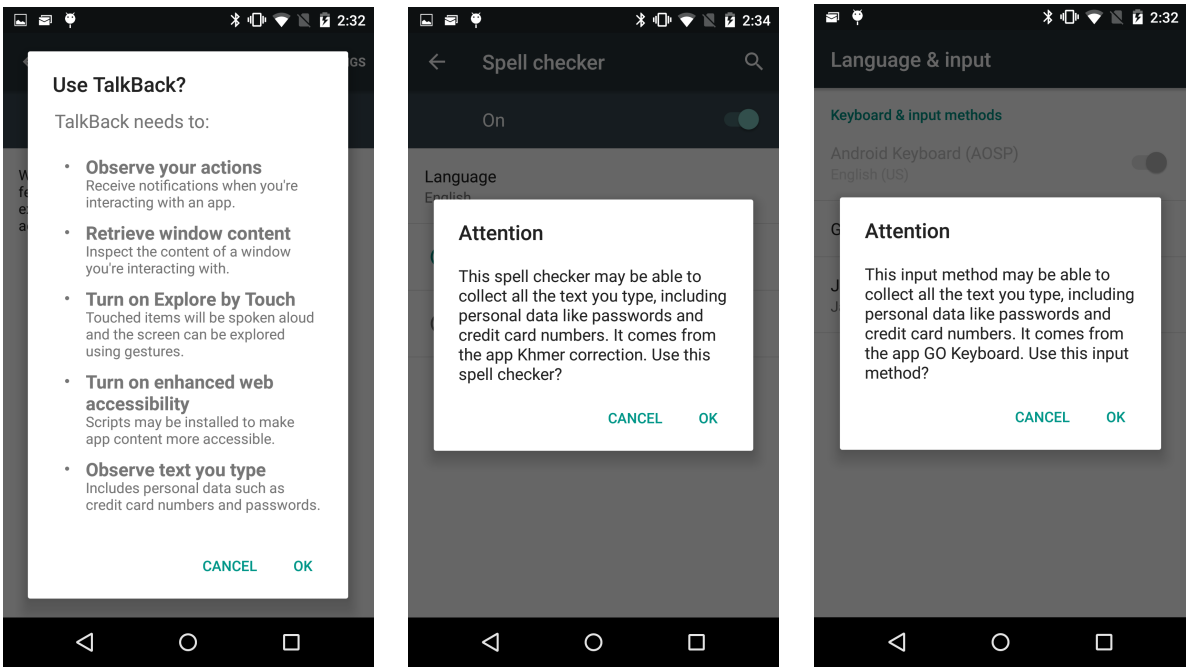
Fig. 4.1.: Design Architecture of Framework Module on Android

the current design of extensible framework on Android confronts the identified security risks in the following section.

4.1 Security Considerations in the Design of Android Extensible Frameworks

4.1.1 Initiator Perspective

From the initiator perspective, Android prevents malicious ones from abusing the provided features (**P1**) with permission requirements. For instance, in order to use the AccountManager module within the target app, the initiator app has to acquire the `GET_ACCOUNTS` permission upon installation. In other cases, the feature provided by the module app will be automatically applied to all installed apps. As an example, all `TextViews` within an app's UI will by default benefit from the spell checking functionality provided by the module app. However, Android does provide mechanisms for the initiator



(a) Accessibility (b) Spell Checker (c) IME

Fig. 4.2.: Consent Screens When Selecting Framework Module App

app to prevent its own resource from being exploited. In the `TextView` case, the initiator app can turn off the suggestions with the `TYPE_TEXT_FLAG_NO_SUGGESTIONS` attribute.

4.1.2 Target Perspective

Due to the sensitivity of the data and functionality exposed to the plugged app, user consent has to be acquired beforehand (**P4**). As shown in Figure 4.2, Android will warn users regarding the potential risks when they select 3rd-party apps to provide accessibility, spell checking or IME functionalities. Since all security-critical features are exposed via a service component, another interesting question is whether that service component can be invoked from other apps. Android prevents this from happening by enforcing a signature-level permission requirement on accessing the service component. Still take the

Spell Checker Framework as an example, the following configuration has to be specified in the module app.

```
<service
  android:name=".SampleSpellCheckerService"
  android:permission="android.permission.BIND_TEXT_SERVICE" >
  <intent-filter >
    <action android:name="android.service.textservice.SpellCheckerService" />
  </intent-filter>
  ......
</service>
```

The `action` attribute within the `intent-filter` tag allows Android to label this app as a module app that provides spelling suggestions. Based on this information, the Settings app can construct the list of spell checker apps installed on the device, and present to the user for selection. To prevent adversaries from intruding into this service component, the module app requires the `BIND_TEXT_SERVICE` permission, which can only be granted to Android framework and preloaded apps. Thus, the target app is resilient to the privilege leakage risk.

4.1.3 Channel Perspective

The binding between Android framework and the plugged app is through Binder IPC. In particular, upon user selection, the responsible system service normally saves the identifier of the selected module app in the Settings database. When handling the request

from the initiator app, the same system service will query the Settings database for the currently effective identifier, and establish the connection with a `bindService` call to the module app. In return, a Binder token will be returned, which allows the system service to access the provided functionality. As the Binder token is dedicated to the `system_server` process, no other parties can eavesdrop this channel (**P2**). Furthermore, adversaries cannot manipulate the binding as the access to the module identifier is protected with the signature-level `WRITE_SECURE_SETTINGS` permission. However, it remains unclear whether the security considerations are extended to cover situations when the integration is finished (**P3**).

Actually, the prosperity of the Android ecosystem contrasts sharply with the short lifespan of apps on devices [1–3, 79–81]. The frequent app uninstallation poses an important security question on the framework module integration scheme: what will happen if an app is uninstalled, but the data being shared are not completely cleaned from the system? Because of the program integration on Android, an app may still depend on the uninstalled one to perform privileged operations. Thus, adversaries can take advantage of such dependency to cause damage. In other cases, adversaries can even forge the identity of the uninstalled apps to steal sensitive data from the device.

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

4.2 Problem Formulation

4.2.1 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 chapter, unless otherwise specified, system services are used 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 in this chapter, 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 (these data are not considered 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.

4.2.2 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 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.

4.2.3 Two-stage Study on Android Data Residue

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. The work presented in this dissertation filled the void by performing a systematic study to reveal the data residue instances in Android and understand their security consequences. The study has excluded the data intentionally left on devices, such as app's backup data and files on

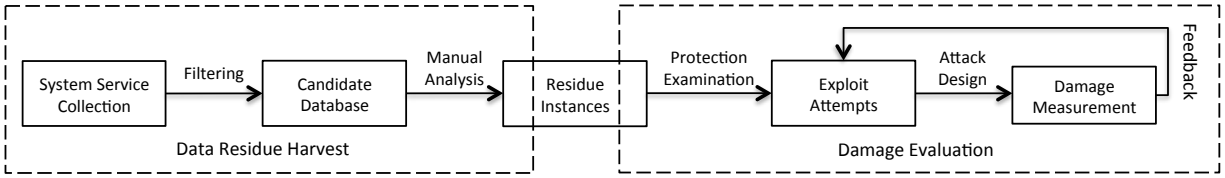


Fig. 4.3.: Methodology of Data Residue Study on Android System Services

the shared external storage. The study consists of two stages: vulnerability identification and automatic detection. This chapter focuses on the manual identification of the data residue vulnerability in one particular Android version, while the details of the automatic data residue detection system are presented in Chapter 5. Without loss of generality, data residue instances refer to the leftover after their owning apps are uninstalled from the device. We call them data residue vulnerabilities if they can be exploited to cause real damages.

4.3 Methodology

The manual identification of the data residue vulnerability consists of two phases: data residue harvest and damage evaluation. Figure 4.3 depicts the flow of the methodology.

Data Residue Harvest To uncover data residues, the study looks at two types of services, i.e., system services and the services in pre-installed apps, because both of them are privileged. All available system services are collected using the *dumpsys* utility provided by the Android Debug Bridge (`adb`). At the same time, pre-installed apps's services are collected by parsing their manifest files. Only the services that are declared as exposed are included in the candidate set (private services are not accessible to other apps).

The source code of these services have been manually inspected to identify data residues. The manual analysis is conducted based on the following two insights. First, 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, the manual inspection focuses 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. The study also systematically examines corner situations that may subvert data cleanup logic, like the `AccountManager` case in Section 4.4.1.

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. The study attempts to unveil these implicit assumptions and more importantly examines their validity. Our study considers three operations that can lead to device state changes: device reboot, app installation and app uninstallation. The experiment design create scenarios to

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 4.1: Worrisome Data Residue Situation on Android System Services

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, real-world attacks are conducted 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.

4.4 Attacks

The study is conducted 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 4.1 summaries the study results. A total of 12 data residue instances are identified, 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, all residue instances are grouped into five categories: *Credential Residue*, *Capability Residue*, *Settings Residue*, *Permissions Residue*, and *History Residue*. For each category, the study examines 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, experiments are designed to demonstrate the feasibility of exploits and show the

potential damage. To make attacks more realistic, as an important principle in the experiment design, declaring suspicious permissions is avoided in the attack apps. Actually, apps with desired capability already exist in various app stores, as shown in Section 4.5, although they are not attempting any attacks simply due to the lack of knowledge on the vulnerabilities discussed in this chapter.

The following subsections will explain the technical details of each attack and the experiment results. For the successful attempts, their preconditions and feasibility in real-world scenarios are further discussed. Besides that, failed attacks are also important pieces in the 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 learned from them are valuable.

4.4.1 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.

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 4.4, 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.

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 4.4, 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.* The experiment is designed to target 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. The observation is 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*

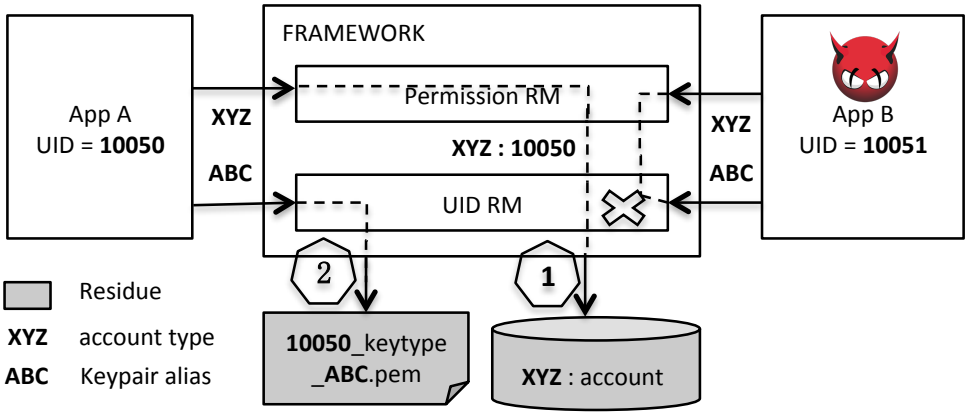


Fig. 4.4.: Android's Protection on Accounts and Keypairs

declare the same account type, and if so, whether that can prevent AccountManager from removing user credentials.

Experiment Design The experiment 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 the 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 the malicious authenticator app will be denied by `AccountManager`'s protection mechanism.

We then uninstalled `myMail`. Surprisingly, Android makes the 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 [82].

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 4.4, 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 [83]. ADAL provides the support for Work Accounts to 3rd-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 the 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 require any permissions from the apps to use the `Keystore` feature, allowing apps to easily hide their malicious intention.

4.4.2 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

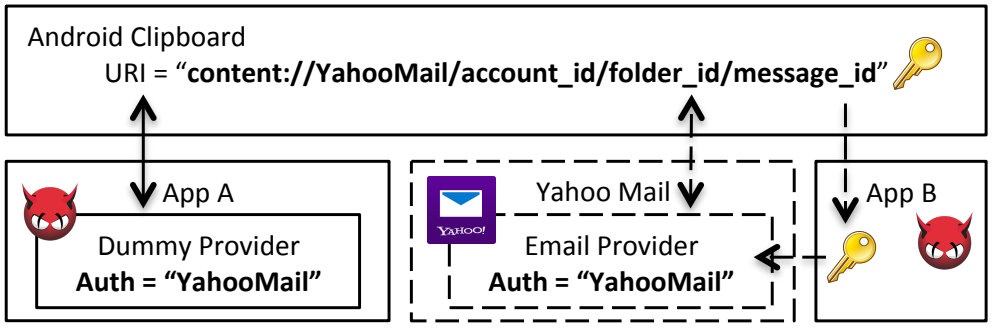


Fig. 4.5.: Yahoo Mailbox Intruding

maintained at any point of time, but not throughout a duration. This fact will be the basis for the 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 GooglePlay. 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.

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 4.5 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 4.5 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. The 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.

4.4.3 Settings Impersonating

As an open platform, Android offers a variety of extensible frameworks for 3rd-party apps to provide system-level functionalities. An example is the Spelling Checker Framework, which can collect user keystrokes and then rely on a 3rd-party app to provide

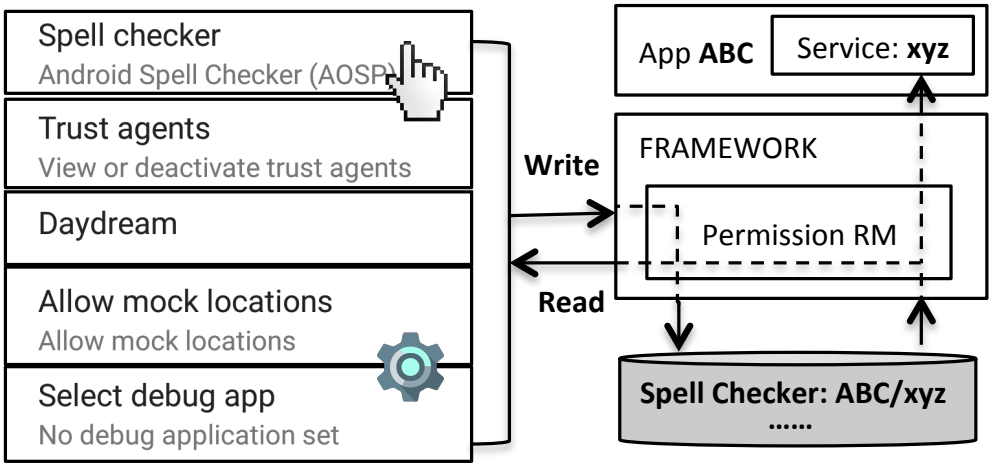


Fig. 4.6.: Android's Protection on Settings Configurations

spelling suggestions. As shown in Figure 4.6, 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 4.6, when the user chooses App ABC as the 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 3rd-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 4.6. 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. We discuss the two most 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, the exploits do not lead to severe damage. 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.

4.4.4 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 4.4.1, the process to steal print content is identical to Settings impersonating attacks in Section 4.4.3.

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 3rd-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.

4.4.5 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 should coexist on the device. Actually, it is quite common for developers to submit multiple apps to the appstore. According to [84] 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.

4.5 Evaluation

In this section, we evaluate how the data residue attacks identified during our manual analysis may potentially affect the real world. To this end, we plan to evaluate (1) the

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 4.2: Impact of Android Data Residue Vulnerability in Practice

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 4.2. 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 [47] 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 4.2(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

	package	account type	authority
GooglePlay	✗	✓	✓
Amazon Appstore	✗	✓	✓
Samsung Appstore	✗	✓	✓

Table 4.3: Assessment on Appstore Defense

attributes. We map out the essential attributes for successful attacks in Table 4.2(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 are summarized in Table 4.3. First, we would like to see whether real-world authenticator apps from the same app market can declare the same account type. This turns out to be true for GooglePlay, Amazon and Samsung appstores. For example, we found that apps from Yahoo and the `AT&T Live` app share the same account type. Another example is the `Contacts+` app and the `Telegram` app, which share the same account type "org.telegram.account". Both instances discussed above probably have legitimate business cooperation reasons, instead of being malicious. However, there is no doubt that the design of `AccountManager` fails to prevent malicious behaviors while preserving the necessary functionality.

Second, we would like to see how app markets deal with duplicate package names and content provider authorities. We submitted our apps to GooglePlay, Amazon and Samsung

appstores. We have observed that if an app with the same package name already exists in the store, our submitted app will be rejected. However, although the uniqueness of package name is preserved on each individual store, it is hard to enforce that across all the stores. For instance, two keyboard apps from Kika can serve as spell checkers, but the package name only exists on GooglePlay. Attackers can freely upload a malicious app with the same package name to Amazon or Samsung appstore. As for the authority attribute, none of the app stores perform uniqueness checks. We are able to upload apps to all three appstores using the same authority name.

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 4.2(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.

4.6 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

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 4.4: Security Examination of Android Attributes Used in Protecting Data Residue

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.

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 4.2(II). Here we further summarize their underlying assumptions, protection effectiveness, and breaking conditions in Table 4.4.

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 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 chapter. 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. Thus, the study in this chapter intends to provide the baseline of the data residue problem, while Chapter 5 presents a more comprehensive understanding of the problem across the entire Android ecosystem.

4.7 Conclusion

In this chapter, 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 the 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.

5. DETECTING DATA RESIDUE IN ANDROID IMAGES

As a pioneer that sheds lights on the data residue vulnerability, our study in Chapter 4 limits its scope to AOSP version 5.0.1 with the requirement of source code and significant manual effort. However, the extensive customization on Android devices from different vendors and high fragmentation of Android operating system demand an automatic, scalable and source code independent framework for data residue detection. In this chapter, we have designed and implemented ANRED[1], an ANdroid REsidue Detector that takes an Android device image as input and quantifies the risk for each identified data residue instance within collected system services. For each examined image, the final output will be a well-formatted report for security analysts to conduct further verification.

The design of ANRED has overcome several challenges. First of all, in order to mitigate the absence of framework source code and limitation of hardware resources, ANRED depends on WALA's static analysis engine to directly work on Java bytecode and performs analysis on the call graph constructed for each system service. In this process, we have utilized several novel techniques to generically preprocess Android images from different vendors, accurately pinpoint all system services within the given image, identify entry points for each system service and connect the broken links on the call graph because of the event-driven nature of Android system. Secondly, the exploits on data residue instances

[1]ANRED is a former French public institution, serving as a national agency for the recovery and disposal of waste.

rely on various triggering conditions, such as device reboot, app installation and uninstallation. They are quite difficult to emulate in static analysis. We have converted this task to a standard relative complement problem in set theory [85], where data residue instances are deduced as the relative complement of the deleting data set with respect to the saving data set. To retrieve those two data sets, we have split the original call graph into two subgraphs originating from the saving and deleting entry points. We have further analyzed the saving logic and deleting logic on corresponding subgraph. While the saving entry points capture all interactions with the apps, deleting entry points are functions that handle app uninstallation. Further complicating the detection process is when the data removal operation is present in Android framework, but the underlying logic is flawed. Without scrutinizing the source code, it is quite difficult to cover these cases. Instead, we have taken the complexity of deleting logic into consideration and quantified the possibility of each detected data residue instance.

We have evaluated ANRED against 606 Android images from all major vendors, such as Google, Samsung, Xiaomi, LG, HTC, CyanogenMod and Moto, covering all platform versions from Gingerbread to the newest Marshmallow. ANRED has successfully analyzed 82.7% of system services within the pre-defined time limit. The risk reports indicate that, there are 191 likely data residue instances on average for each image and 106 (55.5%) of them are missing data deletion logic upon app uninstallation. We have further broken down the results from four perspectives: vendor, version, service category and residue type. We have confirmed that, vendor customization is indeed a major factor in introducing new data residue instances, while the effect of version upgrade varies from vendor to vendor. From the service perspective, the majority (65%) of identified residue instances are from

framework services versus preloaded app services. Moreover, the results have shown that, data leftover are more common in memory data structures and configuration entries. To evaluate the effectiveness of ANRED, we have further examined the risk report generated from ANRED. Specifically, we took the manual analysis result from [4] as the basis, and compared with the risk report from ANRED for the same image. 10 out of 12 instances from previous study are successfully captured in the risk report, while the missed 2 instances are for a previous Android version. In addition, we have identified 5 new data residue instances, leading to data leakage and privilege escalation attacks.

5.1 Background

In this section, we present necessary background knowledge to facilitate the design of ANRED.

Android Image Architecture Android images from different vendors are presented in different formats. However, essentially, they are all compressed files consisting of four major partitions, i.e., `boot.img`, `system.img`, `userdata.img` and `recovery.img`. The most important one is the `system.img`, as it contains Android framework and preloaded apps where all privileged services reside. In Android file system, apps come in packages with the extension `.apk`. Each APK contains app resources, `AndroidManifest` configuration file and code. With the evolution of Android platform, its code format has also been optimized from `.dex` to `.odex` and `.oat` for the purpose of saving storage space and speeding up the booting process. Such complex combinations urge a generic approach to convert various

```
final Class SystemServer{
  public SystemServer(){
    ...
    // starting the AbcService
    ServiceManager.addService
      ("Abc", new AbcService());
    ...
  }
}
```

(a) Service Start-up

```
/* @hide */
interface IAbc{
  ...
  // one of the exposed APIs
  void setComponent(String s);
  ...
}
```

(b) Service Interface

```
Class AbcService extends SystemService
    extends BroadcastReceiver{
  @Override
  onStart(){
    publishBinderService(
        new BinderService());
  }
  Class BinderService extends IAbc.Stub{
    Public void setComponent(String s){
      new MyHandler().sendEmptyMessage();
    }
  }
  Class MyHandler extends Handler{
    @Override
    public void handleMessage(Message m){
      db.putStringForUser("Abc",s);
    }
  }
  @Override
  onReceive(){
    prepare();
    removeData();
    return;
  }
}
```

(c) Service Implementation

Fig. 5.1.: A motivating example that shows the working flow of Android system services and origin of data residue instances.

Android specific code formats into bytecode that is suitable for standard static analysis platforms.

Android System Services Each app on Android runs with a distinct system identity (UID), and by default, does not have permission to perform any operations that would adversely impact other apps, the operating system, or the user. Privileged operations are delegated to certain system services, and apps can request the access by declaring corresponding permissions in their `AndroidManifest` files. At the same time, certain services within preloaded apps can also conduct privileged operations on behalf of the app. Without losing generality, we use system services to refer to privileged services that reside in Android framework as well as in preloaded apps.

All system services on Android execute in the `System_Server` process, and expose their functionalities with APIs defined in corresponding interfaces. Figure 5.1 demonstrates this process with a manually crafted system service `AbcService` based on real instances (`DreamManagerService` and `SpellCheckerService`). In Figure 5.1(a), the `SystemServer` class adds the `AbcService` to the system service list. By doing so, it triggers the lifecycle event `onStart()` in the `AbcService` implementation (Figure 5.1(c)). Upon starting, `AbcService` exposes its functionalities via a Binder object that implements the APIs defined in the `IAbc` interface (Figure 5.1(b)). An interface essentially defines the protocol between two communication endpoints across the process boundary. In this case, an app can use the exposed Binder object to invoke the `setComponent()` API in the `System_Server` process by complying with the protocol defined in the `AbcService` interface. All functions defined in the interface are entry points for outside apps to trigger the saving operations within this privileged system service.

Android App Uninstallation When an app is uninstalled from the Android system, the `PackageManagerService` will remove resources stored in the app's private directories. After that, it sends out a broadcast event to awake the uninstallation handling logic in other parts of the system. In Figure 5.1(c), the `AbcService` will receive such a broadcast event in its `onReceive()` function, and response by removing related data after preparation. Apart from the generic `BroadcastReceiver` approach as seen in Figure 5.1(c), there do exist other ways to get notified upon app uninstallation. To be more specific, `PackageMonitor` and `RegisteredServicesCacheListener` register an uninstallation listener via `BroadcastReceiver`, but further provide easy-to-use APIs to other parties.

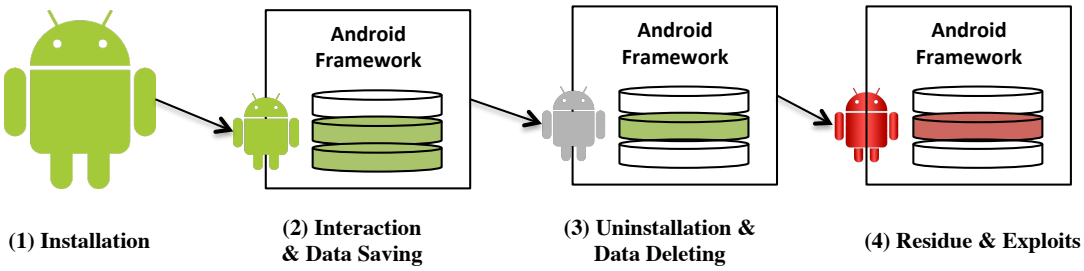


Fig. 5.2.: Flow of data residue generation and exploits on Android

Alternatively, the holder of a Binder object will get a `DeathRecipient` notification when the creator process dies upon app uninstallation. Those four approaches are entry points for triggering data deleting logic within system services upon app uninstallation.

Android Data Residue Vulnerability Data residue exists on Android due to the mismatch between saved data and deleted data within a system service upon app uninstallation. This process usually involves four steps, as illustrated in Figure 5.2. It starts with the installation of a normal app (step 1). Upon user interaction, data related to this app will be saved by certain system services within Android framework (step 2). For instance, the implementation of `AbcService` shown in Figure 5.1(c) asynchronizely handles the IPC invocation of `setComponent()` by saving user configuration into a database with entry key `Abc`. Such asynchronized execution pattern is widely used in Android framework. As system services are meant for serving multiple apps, the data collected from each app are clearly organized based on the owner's identity. This is also necessary for protection; so one app cannot access the data belonging to the other app. Files, databases and well-marked data structures (e.g. `Hashmap`) are normally used to store app-specific data.

Later on, when the app is uninstalled, Android will try to delete related data entries from memory and persistent storage (step 3). In this step, the above-mentioned `Abc` entry will automatically become residue if the app uninstallation handling logic is not in place. However, even if the handling logic is present, there are still chances for the data residue vulnerability to arise. In Figure 5.1(c), the data removing operation happens after executing certain preparation logic. In the case where the preparation logic is too complicated to guarantee the correctness, the entire data removing logic may eventually become invalid, leading to careless data residue instances. Having data leftover does not necessarily lead to security breach, as long as the protection is sound. Otherwise, sensitive information will be leaked out to adversaries(step 4).

5.2 Design

The design of ANRED has experienced three major stages, and in each stage, we have considered multiple design choices.

Technique Vulnerability analysis on Android usually utilizes two major techniques: static analysis and dynamic analysis. In order to mitigate the absence of framework source code and limitation of hardware resources, ANRED depends on static analysis to directly work on Java bytecode. In this process, we have connected the broken links on the call graph with WALA's `shrike` bytecode rewriting utility.

Scope Given the static analysis technique, we have to carefully choose the scope of our analysis. One option would be conducting analysis on the entire framework code base.

However, it has two major drawbacks. First of all, this option will lead to one explosive call graph, and most of nodes on the graph are irrelevant to the data residue vulnerability. Secondly, we also want to detect data residue instances within preloaded apps. Fitting multiple app `jar`s together into the analysis further complicates the situation. The nature of data residue vulnerability has determined that, it can only occur at the level of system services. Thus, it is more viable to conduct static analysis at the same level, i.e., analyzing each system service separately.

Approach System service will be considered as the basic analysis unit in ANRED. Actually, we are interested in two types of operations: saving operation and deleting operation. Mixing both operations on the same call graph will make them indistinguishable at the end. Thus, ANRED builds two call graphs for each system service. The deleting call graph is constructed with app uninstallation handling APIs as entry points, while the saving call graph originates from the interface APIs. Naturally, we analyze a service's deleting logic on its deleting graph and saving logic on its saving graph.

The high-level flow of ANRED is depicted in Figure 5.3. ANRED takes the entire Android image as input, extracts the framework's and preloaded apps' bytecode, and collects candidate system services inside. Although all the bytecode is loaded into the static analysis platform together, each system service is handled separately, i.e., we generate call graphs on a system service basis. For each system service, there will be two call graphs: saving graph and deleting graph. Saving graph captures the control flow and data flow where the data can be saved inside the system service, while deleting graph indicates what data will be removed when handling app uninstallation. The final data

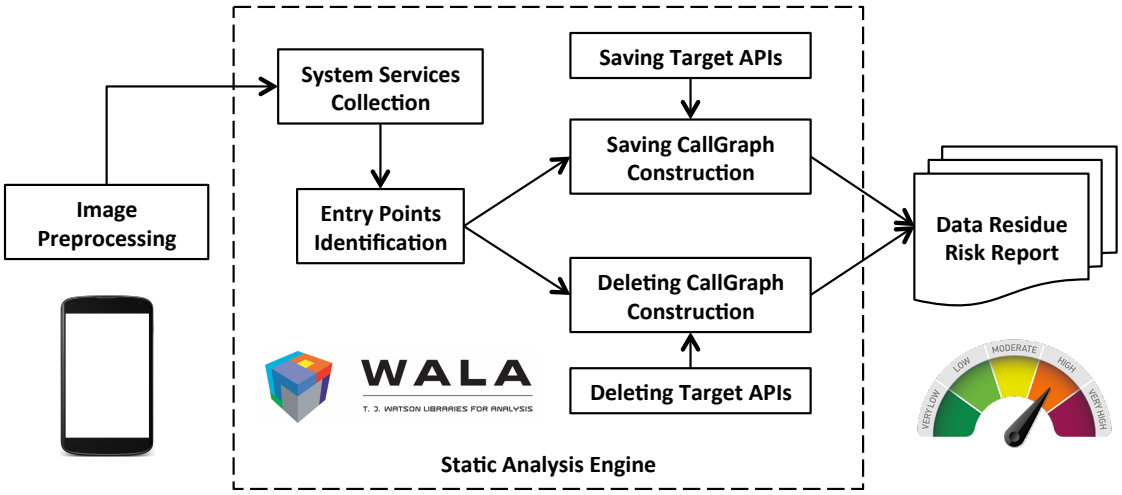


Fig. 5.3.: Overview of ANRED Design

residue risk report is a relative complement of the data removing results from the deleting graph with respect to the data saving results from the saving graph. To cover the careless data residue instances as in Figure 5.1(c), ANRED also takes into consideration the complexity of the deleting logic to quantify the underlying risks. We explain the design details of each block in following sections.

5.2.1 Image Preprocessing

Given an Android image, ANRED extracts the Android framework code and preloaded apps from inside. However, different vendors or different versions of Android pack the code in different formats, ranging from `apk`, `dex`, `odex` to `oat`. ANRED handles all possible combinations gracefully. The `dex2jar` utility can convert `dex` and `apk` files into `jar`s that are suitable for standard static analysis platforms. Additionally, we use `apktool` to retrieve the `AndroidManifest` configuration file from each app. Regarding the `odex` code, `baksmali` and `smali` are capable of converting it to `dex` files. Naturally, we can get the

`jars` with the help of `dex2jar`. Images that contain `oat` files or target at Android Lollipop require special handling with `dextra` [86] and `deodex-lollipop` [87].

5.2.2 System Service Collection

To detect data residue instances, we need to identify all system service classes from a device image. Such list is always available on a running device, but challenging to obtain statically from a compiled image. The reason behind is that, the registration place of each system service varies greatly from image to image and version to version. Instead, ANRED targets at the registration APIs that are more stable across Android customization and version upgrade. The service example shown in Figure 5.1 demonstrates the two most representative APIs, `addService()` and `publishBinderService()`, for publishing a system service. To retrieve the system service list, ANRED first collects functions that invoke `addService()` or `publishBinderService()`. With those functions as entry points, we further construct a call graph. By traversing through the call graph, not only can we pinpoint the places where system services are registered, but also resolve the service class and exposed interface class. For system services that are within preloaded apps, the above process is simplified by directly searching the `AndroidManifest` file for services that are accessible from outside world. In both cases, we further filter out unnecessary `jar`s and generate input specifications for the static analysis platform.

5.2.3 Entry Point Identification

Static analysis on Android platform demands the construction of a precise call graph. For that purpose, an accurate and complete list of entry point functions needs to be provided. In normal Java programs, the only entry point is the `main` function. However, the callback mechanism widely adopted in Android framework complicates this task. In ANRED, we define entry points as asynchronized invocations where their callees are present in the analysis scope, but their callers are not. As shown in Figure 5.1(b), the `AbcService` specifies a list of exposed APIs in the interface file, namely `AIDL`, for apps to interact with. Naturally, all public `AIDL` functions become entry points. However, a great amount of other asynchronized invocation patterns are presented within different system services. One representative example is the `onClick` function inside the `onClickListener` interface. In this case, the system service provides the implementation of `onClick` function (callee), while the caller (user event) triggers the invocation. Clearly, the caller does not exist in the analysis scope, and thus, ANRED considers the callee function, `onClick`, as a entry point.

To collect all entry points, we take a similar approach as in EdgeMiner [88] by searching all internal classes within each system service for interfaces and abstract classes. The reason behind is that, both interfaces and abstract classes rely on other parties to provide the actual implementation, indicating the absence of a caller. Different from EdgeMiner, we exclude functions within `Handler`, `Thread`, `AsyncTask` and `ServiceConnection` classes, since they are asynchronized invocations where the caller and callee are both present in the analysis scope, as illustrated in Figure 5.4(a). Such cases lead to broken links in the constructed call graph, and we will handle them differently in Section 5.2.4. After

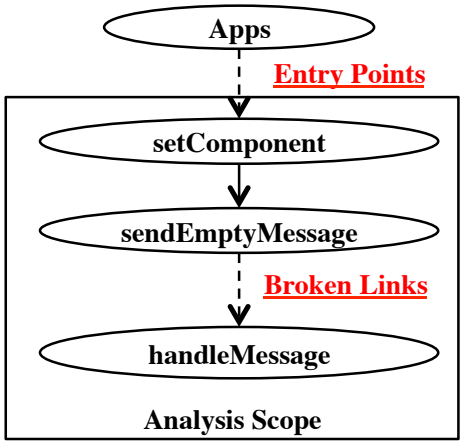
identifying all entry points, we further split them into saving entry points and deleting entry points for the construction of saving call graph and deleting call graph, respectively. We consider all the APIs that handle app uninstallation as deleting entry points, and the rest as saving entry points. Specifically, ANRED includes the following four classes as the source for deleting entry points: `PackageMonitor`, `BroadcastReceiver`, `DeathRecipient` and `RegisteredServicesCacheListener`.

5.2.4 Call Graph Construction

We choose WALA [51] as the static analysis platform for call graph construction, due to its popularity and strength in data flow analysis, which is our main concern in the data residue detection process. The analysis stays at the Android framework level, but inherits all challenges, such as broken links, from the app level. Take the `MyHandler` class in Figure 5.1(c) as an example, the execution is conducted with the help of a `Message` queue. The caller side (`setComponent()`) puts a message on the queue, and waits until it is consumed by the callee side (`handleMessage()`). Both the caller and callee are present in the analysis scope, but the connection is missing. Those broken links greatly affect the code coverage in static analysis. Existing solutions [48, 49, 89] model the behavior of caller and callee functions, and add edges in the constructed call graph to connect them. Such bridges mitigate the reachability issues (control flow) on the call graph, but do not explicitly catenate the data flow. Considering multiple research groups analyzing the same set of `jar` code for different purposes, the effort is duplicated when bridging broken links at the static analysis level.

We would like to connect broken links at the bytecode level, independent from the static analysis platform. The output will be a fixed `jar` file with all broken links connected, and other researchers can apply it directly for different purposes with their own choices of static analysis platforms. ANRED builds upon the `shrike` utility in WALA for the bytecode rewriting. `Shrike` is capable of looping through all instructions from the given bytecode. Ideally, once the invocation that causes broken links is found, we rewrite the instruction to bridge the connection. The real challenges lie in patching a diversity of invocations with a generic scheme. We use the `MyHandler` example in Figure 5.1(b) to explain the details.

First of all, there are various functions for delivering messages with different arguments. Since Java is an object-oriented language, replacing the current invocation to `handleMessage()` requires loading the correct handler instance and constructing the proper arguments. An easier and more stable approach is to add instructions at the end, instead of replacing existing ones. In the `MyHandler` case, ANRED adds the invocation to `handleMessage()` at the end of `setComponent()` function, as demonstrated in Figure 5.4(b). However, adding the instruction still requires the creation of corresponding handler instance and argument instances. To be more specific, in order to invoke `handleMessage()`, an instance of the `MyHandler` class is needed, as well as a `Messsage` instance if the argument is not empty. It is quite challenging to statically accommodate all situations. Our design choice is to invoke the `handleMessage()` API as a static function and provide generic object instances as necessary arguments. Although the generated `jar` may not execute properly, static analysis does not check for conflict method descriptors. Still, we have to resolve the handler class type. The handler class can be inherited, and static analysis will have trouble to accurately identify its class type from the call graph.



(a) Entry Points versus Broken Links

```
1  Class BinderService extends IAbc.Stub{
2    Public void setComponent(String s){
3      new MyHandler().sendEmptyMessage();
4      MyHandler.handleMessage(new Object());
5    }
6  }
7
8  Class MyHandler extends Handler{
9      @Override
10     public void handleMessage(Message m){
11       db.putStringForUser("Abc",s);
12     }
13   }
```

(b) Broken Link - Bridge

Fig. 5.4.: Connecting Broken Links in ANRED via Bytecode Rewriting

Instead, ANRED locates the outermost class, and searches all internal classes for the ones that extend the base handler class or implement the callback interface. Theoretically, such approach will generate false mappings when multiple handler classes are present within the same outer class, however, our observation is that, inside each Android system service, there is usually only one handler class in use. Eventually, after bytecode rewriting, the bridge will be as in Figure 5.4(b).

5.2.5 Target APIs Harvest

To detect mismatches between the storage operations and cleanup logic, a complete list of saving and deleting API pairs is necessary. For instance, the `AbcService` shown in Figure 5.1 uses `db.putStringForUser()` to save entries into a database. To check the existence of data residue, we would like to know the usage of corresponding deleting API (in this case, identical to the saving API) in handling app uninstallation.

The manual identification of such API pairs is a tedious work, and lacks the guarantee on code coverage. Instead, ANRED applies heuristics generalized from manual inspection to harvest API pairs automatically. In particular, we categorize all saving/deleting API pairs into `SQLiteDatabase`, `SharedPreference`, `Settings`, `Java_Util` and `XML`. For each category, we then generalize heuristics at the level of package, class and function. At the top level, each category corresponds to a Java package, and all classes inside will be the source of API pairs. At the second level, we exclude classes that are irrelevant to storage operations. To illustrate, although thousands of classes exist in the `Java_Util` category, we can safely remove those related to `Exception`, `Concurrency` or `Thread`. At the last stage, both saving and deleting APIs follow strict naming schemes, like `put*()`/`remove*()` in `Java_Util` classes and `create*()`/`delete*()` in `File` classes. Such naming schemes allow a further filtering on the API descriptors directly. With those three levels of heuristics, ANRED is able to quickly and reliably retrieve the list of saving and deleting API pairs.

5.2.6 Risk Evaluation

From the steps above, ANRED constructs two call graphs, i.e., saving graph and deleting graph, for each collected system service. Guided by the storage API pairs, we can further gather all saving and deleting operations via traversing through corresponding call graph. Ideally, only the mismatches indicate data residue instances. However, as shown in Figure 5.1 and previous work [4], the existence of data cleanup logic cannot guarantee the correctness of its implementation. As a result, ANRED quantifies the likelihood of each data residue instance with respect to the complexity of data deleting logic. A variant [90]

```
<image carrier="" manufacturer="" model="" name="" region="" version="">
  ......
  <service category="java_util" finished="" ibinder="" name=""
    output_time_cost="" size="" type="">
    ......
    <residue complexity="1000" deletingInstructions="" name=""
      savingInstructions=""/>
    ......
  </service>
  ......
  <serviceDetectionCost></serviceDetectionCost>
  <rewritingCost></rewritingCost>
  <residueDetectionCost></residueDetectionCost>
</image>
```

Fig. 5.5.: Data Residue Risk Report Template

of the standard Cyclomatic complexity [91] is used in ANRED against each deleting function. When the deleting logic is missing, we assign the largest complexity value, indicating the highest risk for this instance to be real data residue vulnerability. For each data residue instance, we also record its saving/deleting instructions and residue type. At the image level, ANRED further aggregates all data residue instances together into a well-formatted XML report, as illustrated in Figure 5.5.The final risk report serves as the basis for security analysts to conduct verification experiments.

5.3 Implementation

In this section, we present the implementation details of ANRED, which makes it a practical, efficient and reliable data residue detection framework on Android.

5.3.1 Bridging Broken Links

ANRED leverages on WALA's `shrike` utility to bridge broken links caused by the `Handler`, `Thread`, `AsyncTask` and `ServiceConnection` classes in Android. We fit `jar` files into `shrike` individually, and traverse through each embedded class, functions in each class and bytecode instructions in each function. For each instruction, we check it against the candidate invocations that can cause broken links. In the `Handler` case, such invocations include `send*Message()` and `sendToTarget()` APIs, while in the `AsyncTask` case, `execute*()` typed APIs will be the candidates. If the examined function does not contain any candidate invocations, we replace each instruction inside with an identical one. Thus, we avoid missing this function in the fixed `jar` file.

For each target instruction identified, we record its position in the function. Right after it, ANRED inserts a static call to the callee function. Such a function invocation requires four types of instructions. `NewInstruction`s are to create generic java objects as function arguments, followed by `StoreInstruction`s and `LoadInstruction`s, which store and load the generated variables, respectively. Eventually, an `InvocationInstruction` initiates the function call. In `shrike`, each `InvocationInstruction` consists of four segments: function descriptor, object type, function name and dispatch mode. As explained in Section 5.2.4, ANRED dispatches this instruction as a static invocation, and searches from the outermost class for the object type. While function name is self-explanatory, an accurate function descriptor could be challenging to obtain. In most cases, we can manually craft the descriptor string according to the Android documentation. However, in the `AsyncTask` case, all callee functions contain `Params...`, known as Java Varargs [92] that takes an

arbitrary number of values with arbitrary types upon invocation. Our observation is that, the concrete argument type will only be presented in the actual implementation of each callee function. Based on that, ANRED overcomes the challenge by directly resolving the function descriptor from the object class implementation. To be more specific, we search the entire class hierarchy for the class that implements the object type, and then obtain descriptor for each function inside.

5.3.2 Building Class Hierarchy and Call Graph

In the data residue detection process, ANRED totally constructs three call graphs: one for system service collection, one for saving analysis and the last one for deleting analysis. One important premise to construct a call graph is to build the class hierarchy. A class hierarchy is the central collection of classes that define the analysis scope. In the system service collection case, the class hierarchy is only for framework `jar`s, while in the latter two cases, we further include preloaded apps. The reason is that, system services within preloaded apps can be obtained directly from parsing apps' `AndroidManifest` files. Those three graphs serve for different purposes, and as a result, demand different level of precision. In particular, we use RTA [93, 94] algorithm in constructing the call graph for system service collection, as it is relatively simple and fast. Also, all we need from this call graph is the identification of certain APIs and resolution of their arguments. On contrast, 0-CFA [95, 96] algorithm is utilized to construct call graphs for saving and deleting analysis, since they both require an accurate control flow and data flow dependency.

On the saving graph and deleting graph, ANRED further resolves the data entry being stored and removed, respectively. For that purpose, we examine all instructions on each call graph node. When an invocation belonging to the target API pairs is found, we resolve the value of its arguments. This is done by consulting the local symbol table and define-use chain. In the case where the argument comes from function input and the current node is an entry point, the data entry can have any value (mark as `*`) depending on the caller side.

5.3.3 Handling Exceptions

Although ANRED attempts to patch broken links gracefully, there are unexpected exceptions thrown in bytecode rewriting. Such exceptions cannot be caught in ANRED's implementation, and once that happens, the entire process will terminate. To avoid this situation, we spawn individual thread for patching each `jar` file within 60 seconds. We monitor the execution status for the current thread every other second, and move to the next one upon interruptions or timeout. Similar technique is also used in analyzing individual system service, as over-complicated ones will cause the explosion in static analysis. Again, we set the default timeout to be 60 seconds for analyzing each service. However, such parameters can be configured differently to satisfy various needs.

5.3.4 Discussion

There are a few limitations coming with ANRED's implementation. First of all, WALA can only perform static analysis on Java bytecode, excluding native Android system services from our study. From the previous study [4], the percentage of native services is

Module	#LOC	Language	Dependency
`Jar` Extraction	1438	Python	`ext4utils` [97]
`Jar` Decompilation	2638	Python	`dex2jar` [98], `apktool` [99], `dextra` [86] `deodex-lollipop` [87], `baksmali`/`smali` [100]
Residue Detection	7568	Java	WALA [51]
Result Analysis	857	Python	-

Table 5.1: ANRED Code Base and Dependency

relatively small. Secondly, the complexity of static analysis has long been a concern. In the data residue detection process, we make the best effort to guarantee the code coverage, but there will be inaccuracy introduced in various stages, such as the code decompilation and broken link re-connection. At last, there will not be binary decisions in the final report, and human effort is still needed to validate against possible data residue instances. However, as shown in Section 5.4.2, the manual involvement has been greatly reduced with the help of ANRED, yet more data residue instances are captured.

5.3.5 Code Base and Availability

The implementation of ANRED consists of around 7,500 **L**ines **O**f Java **C**ode (LOC) and 5,000 lines of Python code with comments included. We further break down the entire code base into four modules: `Jar` extraction, `Jar` decompilation, residue detection and result analysis. The code composition and library dependency for individual module is shown in Table 5.1. To handle different situations, our code base contains functions with small deviations. For instance, we have slightly different Python scripts to extract `jar`s from Nexus images and Samsung images. We emphasize that, those functions are counted twice in the presented statistics, i.e., the count of effective LOC will be smaller. Also, the

	2.3.6	2.3.7	4.0.4	4.1.1	4.1.2	4.2.1	4.2.2	4.3	4.3.1	4.4	4.4.1	4.4.2	4.4.3	4.4.4	5	5.0.1	5.0.2	5.1	5.1.1	6.0.1	Total
Samsung				1	7	1	5	6				40		8					3		71
Amazon														1							1
Blu												1									1
CM		52	15		22		20	1	10		1	11		54			12		92	66	356
Geeks		1																			1
Google	3	1	8	3	5		7	9	1	6		7	7	9	4	4	3	6	49		132
HTC												2				1					3
LG												1			1			1			3
Moto															2						2
Sony							2														2
Xiaomi				4	2		4	2				2		15			5				34
Total	3	54	23	8	36	1	38	18	11	6	1	64	7	87	7	5	20	7	144	66	606

Fig. 5.6.: Vendor and Version Distributions of the Image Collections

entire code base of ANRED, as well as our image collection and evaluation results from Section 5.4, are publicly available on [101].

5.4 Evaluation

Image Collection We have collected a total of 606 Android images from various sources [102–106]. Figure 5.6 depicts the distribution of the image collection with respect to the diversity of vendors as well as coverage of Android versions. In particular, we have covered all major vendors, such as Samsung, HTC, CyanogenMod (CM, in short), Google, Xiaomi, etc, and all major Android versions from Gingerbread up to the newest Marshmallow. Throughout the experiment, we consider the OS provider, instead of the device manufacturer, as the real vendor of the image. For instance, we have labeled Google as the vendor for all Nexus images. CM provides Android OS for various device models from different manufacturers, and thus, it becomes the vendor for all of them.

Experiment Setup We have conducted our experiments on `Dell PowerEdge T620` with `Intel Xeon CPU E5-2660 v2 @ 2.20GHz` running `Ubuntu 14.04.1 LTS`. The `Jar` extraction and decompilation stages took around one week to finish for all images, and the processing outputs are available on [101]. This is a one-time effort, and researchers who want to repeat the experiment do not need to carry out these two steps again. After that, we ran ANRED against each image with 16G heap memory size allocated for the Java Virtual Machine (JVM), and with the default timeout value (60s) for each system service. To reduce the overall time consumption, we have divided our image collection equally into four subsets and run four separate instances of ANRED in parallel. From the results, we have gained a comprehensive understanding of the data residue situation across Android ecosystem. The following sections present our insights, as well as the performance details of ANRED.

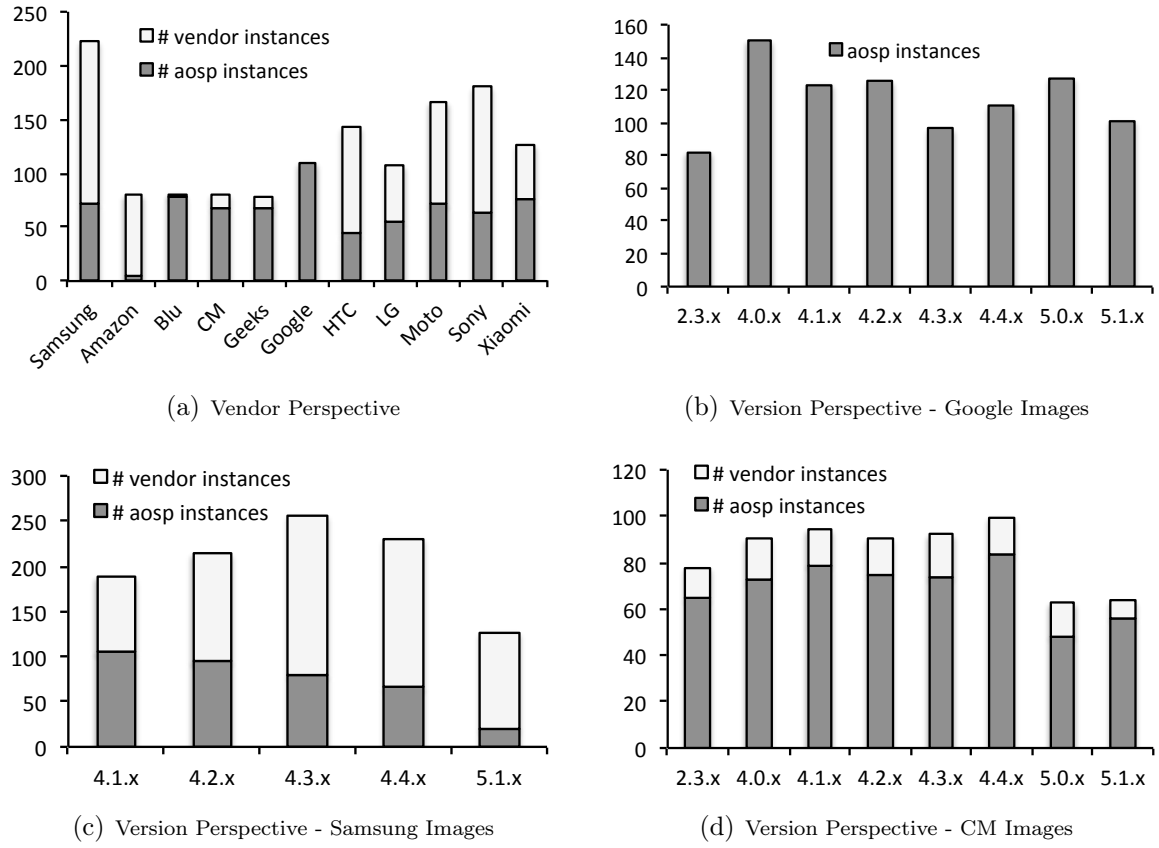
5.4.1 Panorama of Android Data Residue

Overview On average, we have identified 191 likely data residue instances on each image, with 106 (55.5%) of them being labeled as missing data removal logic. Not all of them will necessarily lead to the data residue vulnerability, depending on whether the data leftover is security-critical and exploitable by adversaries. To separate the data residue instances introduced by vendor customization and the ones inherited from the AOSP code base, we consider Google images as the basis. Given one data residue instance on a vendor image, we label it as AOSP instance if such instance has also been identified on Google images with the same version number. Otherwise, we consider it as vendor introduced

instance. For images that do not have a corresponding Google image in our collection, we exclude them from our data analysis. In following sections, we further break down the data residue situation on Android from four perspectives: vendor, version, service category and residue type.

Vendor-wise View In our image collection, the version distribution varies greatly from vendor to vendor. Thus, a direct comparison of average residue count for each vendor will not accurately reflect the effect of vendor customization on the data residue vulnerability. To remove the version bias, we compare the percentage of vendor introduced data residue instances instead. The result is shown in Figure 5.7(a). We have observed that, vendor customization is indeed a big factor contributing to the data residue vulnerability. In particular, vendors like Samsung, Amazon, HTC, and Sony are responsible for more than 65% of data residue instances identified on their images. One extreme case is the Amazon image running 4.4.4. Our analysis result indicates that, 95% of data residue instances identified on this image are due to the heavy customization. Actually, as the level of customization is so high, Amazon renames the OS to FireOS [107]. Even for vendors like Moto, LG and Xiaomi, the percentage of vendor introduced data residue instances is higher than 40%. In comparison, Blu, Geeks and CM make fewer changes to the Android OS, and thus, only introduce a small portion of data residue instances to their images.

Version-wise View Apart from vendor customization, we would like to further evaluate the effect of Android's frequent version upgrade on the data residue vulnerability. The data residue trend across different Android versions is meaningful only if all selected images are



(a) Vendor Perspective

(b) Version Perspective - Google Images

(c) Version Perspective - Samsung Images

(d) Version Perspective - CM Images

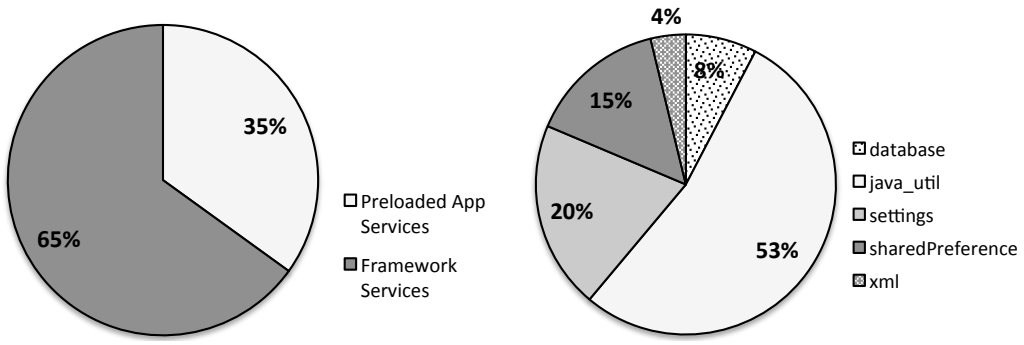
Fig. 5.7.: Effect of Vendor Customization and Version Upgrade on Data Residue

belonging to the same vendor. We have chosen three vendors, i.e., Google, Samsung and CM, because they have images across multiple versions in our collection. The results are depicted in Figure 5.7(b), Figure 5.7(c) and Figure 5.7(d), respectively. As Figure 5.7(b) shows, the average count of AOSP data residue instances fluctuates around 100 across version upgrade. We have observed that, there is always a surge when new Android branches, like Ice Cream Sandwich (4.0.x) and Lollipop (5.0.x), are released. We suspect that, new branches tend to come with new features, and thus, introduce new system services with possible data residue instances.

The version trends on Samsung images and CM images have quite different characteristics, as shown in Figure 5.7(c) and Figure 5.7(d). For Samsung, as the Android version upgrades, we have observed a steady decrease of data residue instances inherited from AOSP code base. In comparison, the percentage of vendor introduced ones rises from 44% to 84%. Our hypothesis is that, Samsung's customization on Android OS has grown heavier as Android evolves, resulting in fewer similarities against AOSP images. Thus, the data residue situation on upcoming Samsung images will be mostly determined by the level of its own customization. On the other hand, the count of data residue instances introduced by CM remains the same at around 15 from Gingerbread to Lollipop. Clearly, the customization performed by CM is quite consistent across version upgrade. In this case, the data residue situation on upcoming CM images depends highly on the corresponding AOSP version release.

Another interesting observation is that, the trends of aosp instances on Samsung images and CM images across different versions are inconsistent with that on Google images. One possible reason is the misclassification of data residue instances. For example, Samsung may have changed the entry names of certain data residue instances coexisting on Google images, but we will not be able to distinguish one another without manually reviewing their source code. We emphasize that, the observations on the data residue situation across vendors and versions are subject to our image collection.

Service-wise View ANRED has considered two categories of system services in the static analysis stage, i.e., preloaded app services and framework services. We have separated their effects, and depicted the result in Figure 5.8(a). As illustrated, the



(a) Service Perspective (b) Residue Perspective

Fig. 5.8.: Effect of Service Category and Residue Type on Data Residue

majority (65%) of identified data residue instances are from framework services. This is consistent with the manual analysis result from previous work [4], where only 2 (download and print) out of 12 instances are within preloaded apps. However, we argue that, the framework developers and preloaded app developers should work together to remove all the data residue instances.

Residue-wise View As mentioned in Section 5.2.5, we have included five types of data residue instances in our analysis, i.e., `SQLiteDatabase`, `SharedPreference`, `Settings`, `Java_Util` and `XML`. With a total of 116K data residue instances identified on all images from our collection, we further show their type distribution in Figure 5.8(b). The result indicates that, 73% of the data leftover are in memory data structures (`Java_Util`) and configuration entries (`Settings`). In comparison, previous work [4] has also identified 8 (2 capability instances, 5 Settings instances and 1 permission instance) out of 12 instances belonging to those two types. It is worth mentioning that, among all data residue instances, 1,629 (1.4%) unique ones are found. In this process, we have used a combination

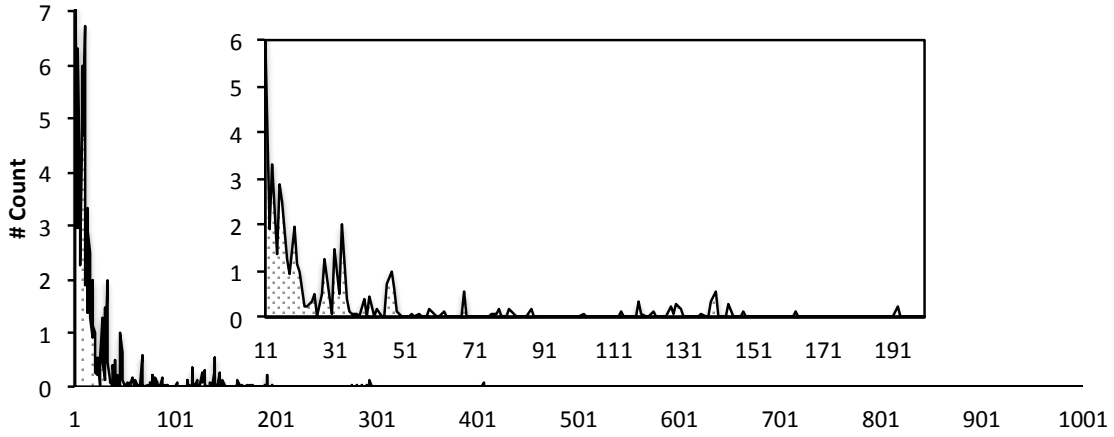


Fig. 5.9.: Data Residue Instance Distribution based on the Complexity of Deleting Logic

of service name, residue type and entry name as the key to remove duplicates. Moreover, only 312 (19%) of the unique ones are from AOSP images, while the rest (81%) are all introduced by vendor customization.

Risk Quantification In our analysis, we have identified 85 instances on each image that have data deletion operations in place upon app uninstallation. However, the complexity of each one's deleting logic varies greatly. We further calculate the average count of data residue instances for each complexity value. The overall distribution is shown in Figure 5.9. A total of 42 (50%) instances have deleting complexity value less than 10, and thus, should be considered as safe. However, the deleting logic of the other half is overcomplicated and may lead to security flaws. To understand how bad the situation is, we zoom in to the region with complexity value between 11 and 200, as shown in Figure 5.9. Surprisingly, a significant portion of functions that handle data removal upon app uninstallation even have complexity value larger than 30. Based on our analysis, we suggest system service developers to follow a clear guideline to remove app data upon its uninstallation.

5.4.2 ANRED Effectiveness

To demonstrate the effectiveness of ANRED, we have evaluated it against the AOSP 5.0.1 image, which was manually examined in previous work [4]. We use their results as the basis for comparison. The analysis takes 11 minutes in ANRED with the default 60s timeout value, and covers a total of 133 system services. Among them, 123 (%92.5) services are finished within timeout limit. Totally, 253 likely data residue instances are identified on this image. More importantly, 205 (%81) of them have a complexity value larger than 10, which are considered as higly risky [90]. We have further validated all 205 risky instances manually. This process was completed by one single Android security analyst within a day, in comparison with 6 person-month in the previous work [4]. It is made possible because of ANRED's detailed report, which not only presents each likely data residue instance, but also pinpoints its saving instruction and risky deleting instruction. We envision that, all Android vendors will utilize ANRED in a similar manner to remove data residue instances from their images before the final release.

Despite the significantly reduced human involvement, we are able to capture 10 out of 12 real data residue instances presented in [4]. The only two missing instances are `Keystore` and `Download`, both of which have been resolved on Android Lollipop and the above-mentioned work reproduces the vulnerability on KitKat and prior versions. One interesting case is the `spell_checker` residue, which is caused by flawed data deleting logic. Although ANRED picks up the instance successfully, the complexity value is only 2. Thus, it has been excluded from our manual analysis. However, ANRED confidently identified a related instance, i.e., `spell_checker_enabled`, which indicates whether the

Instances	Category	Damage	Frequency
backup_transport	Settings	data leakage	359 (%59)
app restrictions	XML	privilege escalation	396 (%65)
notification_policy	XML	privilege escalation	200 (%33)
app-ops	XML	-	-
media_store	Content Provider	-	-

Table 5.2: New Data Residue Instances Identified by ANRED on AOSP 5.0.1

spell checking functionality is enabled or not. From validating this instance, the analyst actually found the previously excluded `spell_checker` data residue instance. Another representative data residue vulnerability is on Android printing framework. ANRED actually identifies two related instances: one from the `XML` category corresponding to the print record residue and the other one from the `File` category mapping to the print content residue. In additional to that, ANRED has also found five new data residue instances. We further present each vulnerability and exploit details in Section 5.4.3.

5.4.3 Attacks

We have conducted manual validation on the analysis result for AOSP image 5.0.1, and found five new residue instances. Three of them are actually exploitable, leading to data leakage and privilege escalation attacks. We have further measured the frequency of those three vulnerabilities within our image collection. Table 5.2 summaries our findings. The result indicates that, such vulnerabilities are pervasive, with a total of 571 (%94) unique images containing at least one of them. Now, we explain the details of each instance.

Backup Mis-transport Android backup framework helps to copy a user's persistent app data to remote cloud storage. Internally, there is a system service called

`BackupManagerService`, which forwards all requests from registered client apps to the current enabled backup service. Android backup service serves as a backup transport between the device and the remote storage. Once enabled, the package name and service name will be jointly saved into Android Settings with entry name `backup_transport`. However, this configuration entry will remain effective even after the referring app has been uninstalled. As a result, an adversary can impersonate the uninstalled app and mis-transport the user's private data to malicious servers. This is not a big concern right now, as only vendor issued backup services can be installed on the device. But in the future, if Android decides to open this feature to 3rd-party apps, such a residue instance demands great attention.

App No-restrictions Since the introduction on KitKat, the multi-user feature on Android has been a viable approach for creating another restricted environment on the same device. On tablet devices, it is called restricted profile and has been used mainly for parental control. On Android phones, such restricted environment, namely managed profile, is favored by enterprise companies to control the environment where company-specific apps and data are running. In both cases, there is an administrator app responsible for specifying the privilege of individual apps. Such configurations will be saved into `user_id.xml` by a system service, `UserManagerService`, with attribute `restrictions`. However, we have found that, these access control policies will not be removed, even after the targeting app has been uninstalled. In return, an adversary can inherit all privileges from the uninstalled app.

Notification Flooding Android supports three levels of notification restrictions on each app, i.e., allow, block and priority. By default, any notifications from the app will be allowed. However, a user can flip the configuration from the Settings app or choose to show priority ones only. The corresponding specifications will be saved in `notification_policy.xml` by a system service, namely `NotificationManagerService`, with attribute `notification-policy`. ANRED red-flaged this instance with the highest risk, and our manual verification has confirmed that. To be more specific, when an app is removed from the device, its notification policy configured by the user will be left over. Thus, an adversary can impersonate this app and flood the user with annoying notifications.

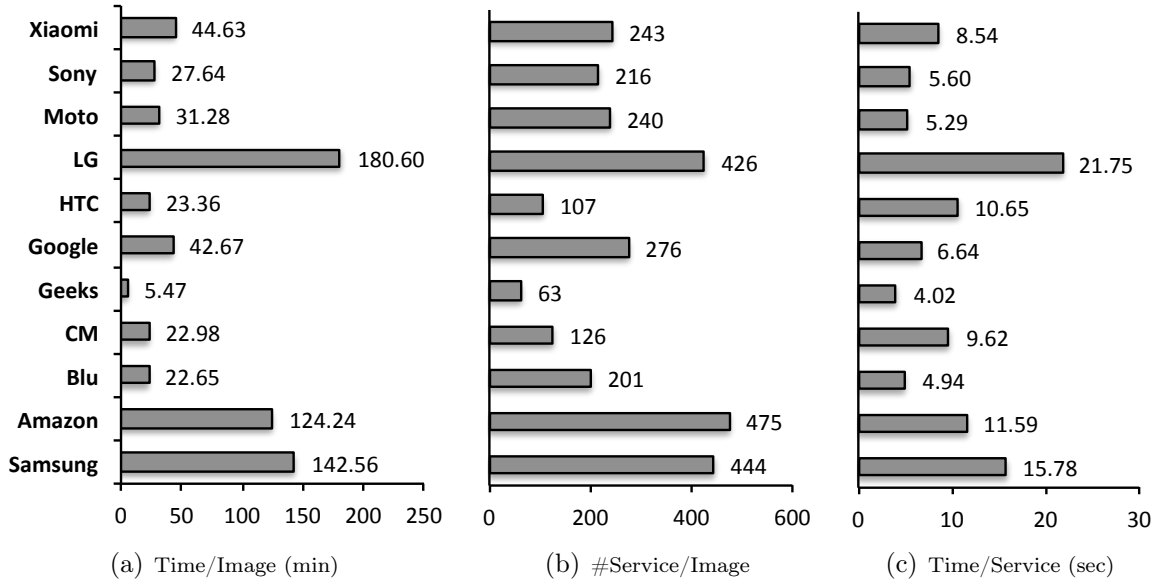
Dawdling Cleanup Another two data residue instances have been found in our manual verification, but with limited implications. For instance, Android system service, `AppOpsService`, saves restrictions on app operations into `appops.xml`, but fails to immediately clean them up upon app uninstallation. Actually, the cleanup task is scheduled periodically for every 30 minutes. The second instance is related to the `mediastore` provider on Android, which contains meta data for all available media on both internal and external storage. An app can send a request to `MediaScannerService` for adding a media file into the `mediastore`. When the app is uninstalled, although the file is deleted from the file system, its meta data remains in the `mediastore` until Android scans the system for new media, which typically happens when the system first boots up or can be called explicitly from apps. Noting that the meta data includes a URI referring to the actual file, we further evaluated it against the capability intruding attack as in the

Clipboard residue case [4]. However, as it turns out, URIs saved in `mediastore` do not possess any capabilities and immune from this attack. In both residue cases, although the attacking time window is quite small, we argue that, a timely data cleanup approach, like `BroadcastReceiver`, should be employed to provide better security guarantee.

5.4.4 ANRED Performance

Time Consumption Because of the parallel execution of four ANRED instances, we managed to finish our experiments in 6 days. On average, it takes ANRED 43.5 minutes to analyze one image, with 1.6 minutes (3.6%) on system service extraction, 6.5 minutes (15%) on `jar` patching and 35.4 minutes (81.4%) on static analysis. However, such time varies greatly from vendor to vendor, as shown in Figure 5.10(a). The most significant factor contributing to the time difference is the level of customization. Actually, images from most vendors take less time than average, with exceptions of Samsung, Amazon and LG, which are known for the heavy customization on their Android devices.

During the customization, vendors usually introduce new system services into Android framework as well as preloaded apps. We have measured the average number of system services analyzed in ANRED for different vendors. Figure 5.10(b) depicts the result. Unsurprisingly, ANRED picks up the largest number of system services from Samsung, Amazon and LG images. Interestingly, Nexus devices from Google rank fifth in this measurement, even though they are equipped with nearly vanilla Android OS. One possible reason is that, Nexus devices tend to preload an entire bundle of Google apps, leading to explosive number of system app services. Furthermore, the level of customization is also

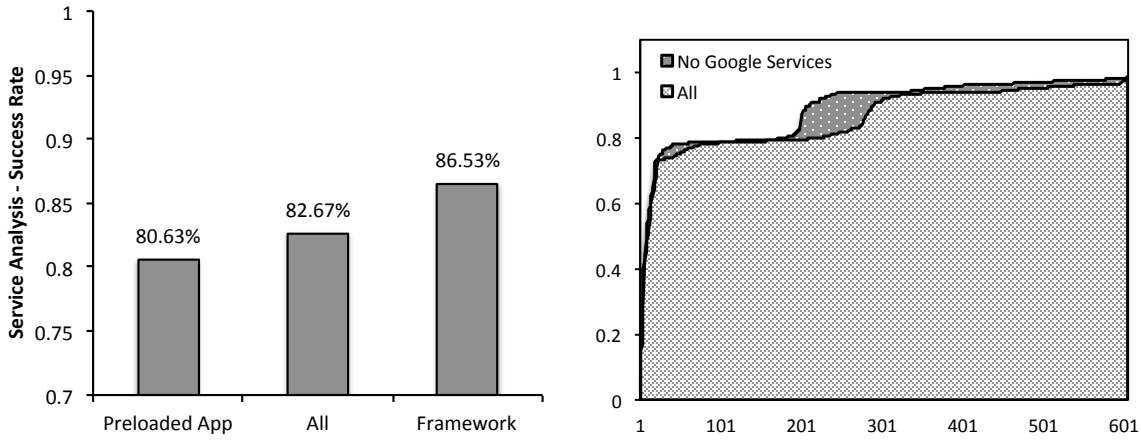


(a) Time/Image (min) (b) #Service/Image (c) Time/Service (sec)

Fig. 5.10.: ANRED Efficiency Breakdown

reflected in the complexity of system services. Intuitively, system services with smaller code size will take less time to analyze in ANRED. Figure 5.10(c) breaks down the average service analysis time for different vendors. Again, Samsung, Amazon and LG images are shown to have more complicated system services than others.

System Service Analysis - Success Rate With an average of 205 system services being analyzed for each image in ANRED, 82.67% of them are finished in our experiment. To understand the reason behind analysis failure, we first separate framework services from preloaded app services, and further measure their success rates. The result is presented in Figure 5.11. It is clear that, framework services (86.53%) have a higher success rate than preloaded app services (80.63%). A closer look at the `jar` decompilation results reveals that, certain preloaded apps have applied code obfuscation technique in the final packaging



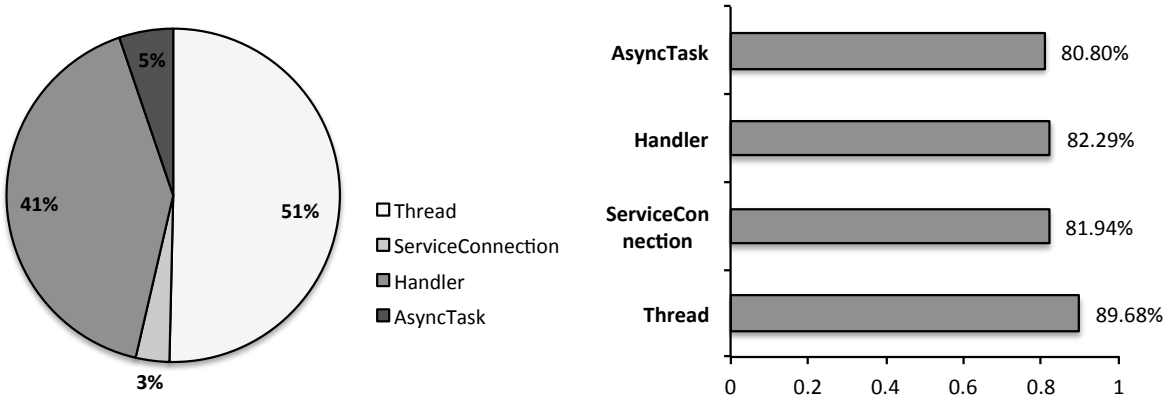
(a) Framework v.s Preloaded App (b) Obfuscation Effect

Fig. 5.11.: Understanding the Success Rate of ANRED's System Service Analysis

process. The decompilation utilities used in ANRED failed quietly on those obfuscated `jar`s, leading to exceptions in the static analysis stage.

To measure the effect of code obfuscation, we calculate the success rate again excluding all system services from Google apps, whose code are commonly obfuscated. The new success rate increases to 88.11% (vs. 82.67%). In addition to that, we have further examined the obfuscation effect on each individual image, as shown in Figure 5.11(b). We have indexed each image based on its success rate in ascending order, then visualized the success rate difference between with and without services from Google apps. As Figure 5.11(b) shows, the success rate of nearly all images benefit greatly from the exclusion of obfuscated `jar`s.

Another factor affecting the success rate is the timeout value. In the experiment, we have used the default 60s timeout value. With more time allowed for each service, the success rate of ANRED's system service analysis will further increase. We argue that, the success rate yields no direct connection with ANRED's fundamental design. Instead, it is



(a) Broken Links Distribution (b) Accurate Connection Rate

Fig. 5.12.: Statistics on Bridging Broken Links in ANRED

an end result of limitations on existing technique (decompilation of obfuscated code) and resources (available servers).

Broken Link Patching Stats In our experiment, ANRED totally patched around 3 million broken links. Figure 5.12(a) shows their distribution in each category. In particular, `Thread` and `Handler` are the two most commonly used classes in Android framework and preloaded apps. They account for a total of 92% of the overall patched broken links. As mentioned in Section 5.2.4, ANRED patches broken links with static invocations inserted at the end. To accurately resolve the object type, ANRED depends on the heuristic that, inside each Android system service, there usually exist only one implementation for `Thread`, `Handler`, `ServiceConnection` and `AsyncTask`. When this heuristic does not hold on certain system services, ANRED falls back to consider the base class as the object type. We have measured the accurate connection rate of ANRED for different categories of broken links. The result in Figure 5.12(b) demonstrates the reliability of ANRED with 86% accurate connection rate on average. The patching accuracy for `AsyncTask` and

`ServiceConnection` is slightly lower, indicating that multiple implementations of them may coexist in certain system services. Even for those cases, ANRED is still able to patch broken links conservatively without causing any exceptions.

5.5 Conclusion

In this chapter, we have designed and implemented ANRED to automatically detect data residue instances on a large scale of Android images with minimal human involvement. The evaluation results against 606 images have again brought questions over the extensive vendor customization and frequent version upgrade on Android. We hope that, vendors can use ANRED to check their images against the data residue vulnerability before shipping with new devices. More importantly, Google should take the lead to provide a clear guideline in reacting to the event of app uninstallation. Additional efforts are also required from the research committee to propose a runtime solution to eliminate the data residue vulnerability on the fly.

6. AFRAME: ISOLATING ADVERTISEMENTS FROM MOBILE APPLICATIONS IN ANDROID

To restrict access to private personal data and privileged phone functions, Android defines a permission-based security model. Apps statically declare the permissions they require, and the Android system prompts the user for consent at the time when the app is installed. Upon approval, the app will be installed with a unique User ID(UID), and be granted the declared permissions. When the app tries to perform some privileged operations during runtime, such as access to the Internet, access to coarse or fine location information, or read the Contact lists, the Android system will use the app's unique UID to find out its granted permissions and then perform access control.

In Android, permissions are assigned at the app level, so all components in the same app have the same privileges. This is not a problem if all these components come from the same developers. However, this is not the case in Android and most mobile systems. In these systems, apps often have third-party components. The current integration scheme to embed 3rd-party code into host apps cannot separate privileges, resulting in over-privileged problems.

In some situations, apps may need fewer permissions than 3rd-party code. To allow 3rd-party code to run, apps have to request more permissions than what they actually need, leading to over-privileged apps. In other situations, apps may need more permissions than 3rd-party code; when users grant the permissions to the apps, they also grant the

same permissions to the 3rd-party code, leading to over-privileged 3rd-party code. Such a problematic integration scheme has led to a series of security risks (**P1**, **P2**, and **P4**). In particular, existing literature has shown that, over-privileged advertising components tend to aggressively harvest users' privacy [5, 6]. Several ideas, such as AdDroid [22] and AdSplit [23], have been proposed to solve these problems. However, both solutions fall short in balancing the desired security against real-world practicality.

The AFrame work presented in this chapter targets at this problem, and relies on a special characteristic observed during the integration of advertisements and apps: most advertisements do not interact with their hosting apps, i.e., advertisements and apps essentially execute in mutual isolation [108, 109]. This special characteristic enables us to totally isolate them from host apps, i.e., running each in a separate process, and even with a different user ID. If such isolation can be achieved, we can directly use the access control system in Android and its underlying Linux to enforce privilege restriction, by assigning different permissions to different UIDs.

The isolation approach was first attempted by AdSplit, but it provides an emulated solution. In AdSplit, an advertisement and its host app are split into two different activities (and can thus be executed in two different processes), with the advertisement activity being put beneath the app activity. Using the transparency technique, AdSplit allows users to see the advertisement through the transparent region in the app activity. However, the use of the transparency technique can be problematic. First, transparency has been widely used by the ClickJacking attack and its variations [24]. Using it for every app (with advertisement) may make the attack detection and countermeasures difficult. Second, transparency poses a significant overhead in drawing, as it requires multiple layers

of drawing surfaces to be combined. On the other hand, AdSplit changes the current mobile advertising architecture by requiring a stub library inside each app to package up requests from advertisement activities and pass them onto their newly introduced advertisement service [23]. Such a stub library is responsible for supporting all the same APIs from the original advertising SDK, but transferring them into IPC function calls. Even though the stub library is straightforward to implement, as they argued in the paper, it is quite challenging, if not impossible, to cover all the existing APIs from the current advertising SDKs. Automated tools mitigate the challenges to some extent, but their commercial implementation will require significant testing effort and introduce new corner cases [23], which lowers the possibility of adopting AdSplit.

A non-emulated solution is desired without using the transparency technique or changing the original advertising architecture, where advertisements and apps are placed on the same drawing surface, but they are executed in different processes. The inspiration comes from browser's `iframe`, which allows a web page to embed another web page, and these two pages are isolated if they come from different origins. Iframe is widely adopted, because of its isolation and easy-to-use properties. A similar "frame" can be supported in activity, allowing an activity to embed another "activity". From the user perspective, these two activities look like one because they seamlessly appear on the same window and behave like one unit. However, from the system perspective, they actually run in two different processes with different user IDs. We call such a frame *AFrame* ("Activity Frame").

AFrame achieves the process/permission isolation that is also achieved by AdSplit, but it goes beyond that by providing display and input isolation. For display isolation, each of those two activities occupies a part of the screen and one cannot tamper with the display

of the other. For input isolation, the host app does not have any information about the user interaction targeted at the AFrame region and vice versa. No activity can inject an event to the other activity either. Another important advantage of AFrame is the fact that using AFrame, developers can use the advertising SDK like before, all they need to do is to tell Android to load the advertising code in a special region. In AdSplit, developers need to replace the original advertising library with their own stub library and override all the public methods. They also need to construct a standard Android IPC message in order for the stub library to communicate with the AdSplit advertising service. Although this process can be automated to some extent, such changes make it difficult to extend the AdSplit to isolate other 3rd-party components. Using AFrame, developers can simply place advertisements and any untrusted 3rd-party component in an AFrame region with very minimal efforts.

6.1 Overview of AFrame

The main objective of AFrame is to isolate untrusted 3rd-party code, such as advertising code, from its hosting app. If the code does not have a user interface (UI), the isolation can be easily achieved. What makes the problem interesting is the UI.

In Android, an app's window that interacts with users is called an *activity*, and its corresponding Java class must be a subclass of the *Activity* class. The idea is to embed another activity inside an activity. From the user perspective, these two activities look like one because they seamlessly appear on the same window and behave like one unit. However, from the system perspective, these are two activities, running in two different

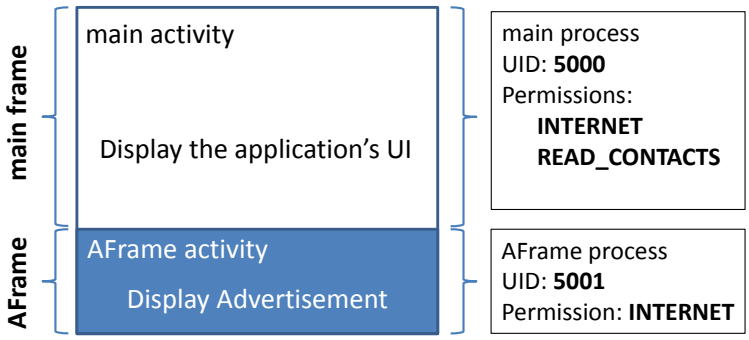


Fig. 6.1.: AFrame Example

processes with different user IDs. We call such a frame *AFrame* ("Activity Frame"), and the hosting app's frame *main frame*. We call the "activity" inside AFrame the *AFrame activity*, and the one hosting the AFrame the *main activity*. AFrame is like a typical *View* component; it occupies a rectangle area in the main activity. Inside this area runs another process, the *AFrame process*. The process that runs the main activity is called the *main process*. Figure 6.1 gives an example of an activity with an embedded AFrame.

Before discussing the design of AFrame, its objectives are presented from three perspectives: user, developer, and system.

6.1.1 From the User Perspective

AFrame should be transparent to end users, who should feel that they are interacting with one activity, not multiple activities. This implies that the lifecycle states and the visibility of AFrame activities and the main activity should be perfectly synchronized. When one goes to the background, the other cannot stay on the foreground; when the main process is killed by user navigation or system, AFrame processes should be killed as well. It should be noted, when AFrame is killed, we do not necessarily need to kill the main

```
Step 1:  public class main_activity extends Activity {
           public void onCreate(Bundle bundle) {
             super.onCreate(bundle);
Step 2:      setContentView(R.layout.main_layout);
             // Look up the AdView and load ads
Step 3:      AdView adView
                 = (AdView)findViewById(R.id.adView);
Step 4:      adView.loadAd(new AdRequest());
         } }
```

(a) Without AFrame

```
Step 1:  public class main_activity extends Activity {
           public void onCreate(Bundle bundle) {
             super.onCreate(bundle);
Step 2:      setContentView(R.layout.main_layout);
         } }
```

```
Step 1:  public class aframe_activity extends Activity {
           public void onCreate(Bundle bundle) {
             super.onCreate(bundle);
Step 2:      setContentView(R.layout.aframe_layout);
             // Look up the AdView and load ads
Step 3:      AdView adView
                 = (AdView)findViewById(R.id.adView);
Step 4:      adView.loadAd(new AdRequest());
         } }
```

(b) With AFrame

Fig. 6.2.: Activity Code Without AFrame and With AFrame

activity, especially if AFrame is used to display advertisement; therefore, even if the advertisement crashes due to a bug or exception, only the AFrame process will terminate, the main activity can continue without the advertisement.

6.1.2 From the Developer Perspective

AFrame should also be transparent to advertising SDK developers, who may not even be aware of the existence of AFrame. No additional modifications or components are required by AFrame in the current or future SDK releases. However, to use AFrame, app developers do need to make minimal changes to their code. The rest of this subsection

focuses on how developers can use AFrame to reduce the risks caused by 3rd-party code. AdMob is used as an example of 3rd-party code.

In Android, to include AdMob in an app, four steps are typically involved (see Figure 6.2(a)). Step 1 defines an activity; Step 2 configures the UI based on a layout file specified by the developer. If there is no advertisement, Steps 1 and 2 are sufficient. To include an advertisement, two more steps are added: Step 3 finds the UI component that will be used for displaying the advertisement, and Step 4 loads the advertisement.

To use AFrame, developers need to split the original activity into two: one is the main activity, and the other is the AFrame activity. Their code is depicted in Figure 6.2(b). For the main activity, only the first two steps are needed, because there is no need to load the advertising code. For the AFrame activity, in addition to the essential Steps 1 and 2, it needs to load the advertisement using Steps 3 and 4. The code for the AFrame activity is quite similar to the code for the original main activity, except that the layout files used are different.

The User Interface. The visual structure for user interface (UI) is called *layout* in Android. Layout elements can be declared in two ways: (1) declare UI elements in XML, and (2) instantiate layout elements at runtime. The AFrame implementation supports both of them, i.e., defining the content of AFrame region statically in the AFrame layout file or dynamically in its activity code. The static case is used as an example in this section.

In Android, an activity has a layout file; in the AFrame design, it has two layout files: one for the main activity, and the other for the AFrame activity. In the main activity's layout file (Figure 6.3(a)), it reserves a place for AFrame using the introduced tag called

```
<LinearLayout>
  <!-- main_activity's display region -->
  <LinearLayout
      android:height="fill_parent"
      android:width="fill_parent">
  </LinearLayout>
  <!-- reserve region for AFrame -->
  <AframeReserve
      android:height="50dip"
      android:width="fill_parent"
      android:layout_alignParentBottom="true">
  </AframeReserve>
</LinearLayout>
```

(a) main_activity Layout

```
<LinearLayout>
  <!-- AdMob advertisement-->
  <com.google.ads.AdView
      android:id="@+id/adView"
      android:height="fill_parent"
      android:width="fill_parent">
  </com.google.ads.AdView>
</LinearLayout>
```

(b) aframe_activity Layout

```
<manifest ... >
  <!-- application permission requests -->
  <uses-permission
    android:name="android.permission.INTERNET" />
  <uses-permission
    android:name="android.permission.READ_CONTACTS" />
  <application ... >
    <!-- main_activity declaration-->
    <activity android:name=".main_activity" ... />
    <!-- aframe_activity declaration & permission requests -->
    <aframe android:name=".aframe_activity" >
      <aframe-permission
            android:name="android.permission.INTERNET" />
     </aframe>
  </application>
</manifest>
```

(c) Application Manifest

Fig. 6.3.: Development Details of AFrame Example

`AframeReserve`. In Figure 6.3(a), the main activity reserves the bottom 50dip region for AFrame to display the advertisement.

The main activity only specifies an empty container for AFrame, without specifying any visual layout inside AFrame. The AFrame's layout is specified in a separate layout file (Figure 6.3(b)). Because only AdMob code is inside the AFrame, the entire layout only consists of AdView, which is the visual component used by AdMob.

The Permission Assignments. In Android, each app is granted a set of permissions during its installation. In the AFrame design, the activity running inside an AFrame has its own permission set, which can be different from the one granted to its hosting app. App's permissions are specified by the tag `uses-permission` in the manifest file named `AndroidManifest.xml`. A new tag called `aframe` has been introduced, and in this tag, developers can specify the configuration for an AFrame, including its associated activity and the permissions granted to it. In the example depicted in Figure 6.3(c), the AFrame is only granted the *INTERNET* permission, while the app has the *INTERNET* and *READ_CONTACTS* permissions.

6.1.3 From the System Perspective

The main objective of the design is to isolate AFrame activities from the main activity. The isolation objective consists of four concrete goals: process isolation, privilege isolation, display isolation, and input isolation.

The first goal is *process isolation*. The AFrame activity and the main activity will run in two different processes with different user IDs. This is a strong isolation, and if it can be achieved, the underlying operating system (Android uses Linux) can be used to protect each activity's memory space, data, files, and other resources from one another. Even if the untrusted 3rd-party code has malicious native code, the isolation will not be broken, unless the 3rd-party code can root the phone.

The second goal is *permission isolation*. Android's permission system is based on user ID, i.e., permissions are assigned to individual user IDs, each representing an app. Because

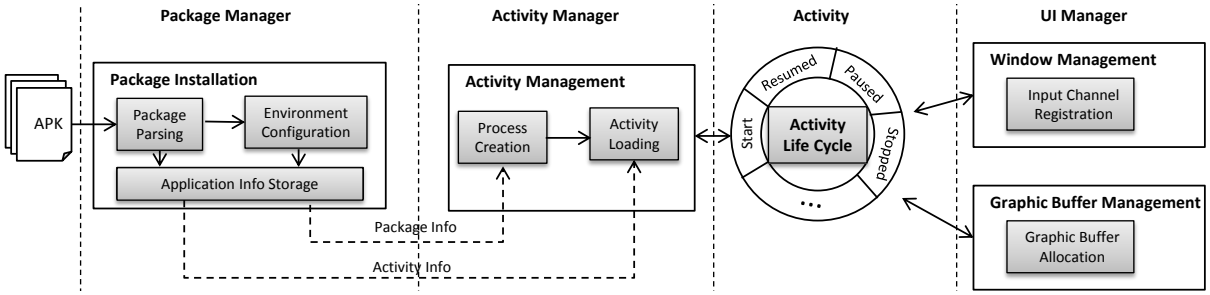


Fig. 6.4.: Activity Creation and Execution

AFrame runs as a separate process with a different user ID, we can naturally use Android's permission system to give the AFrame activity permissions that are different from those given to the app.

The third goal is *input isolation*. When users interact with an AFrame activity, the main activity should not be able to observe this interaction, and vice versa. The interaction includes the inputs occurred in the AFrame or the main frame regions, such as touch events and key strokes. Moreover, AFrame activity and main activity should not be able to forge an event to one another.

The last goal is *display (output) isolation*. Although AFrame activities and the main activity share the same screen, and behave like one activity, they have their own display regions. One cannot tamper with the contents displayed in another activity's region, i.e., their drawings should be restricted to their own regions on the screen.

6.2 Design and Implementation

To achieve the above goals in Android, we need to understand how exactly an activity is created, and how it interacts with the rest of the system. Based on this understanding, several places need to be changed in order to support AFrame. Figure 6.4 depicts the

process of an activity from application installation, activity launch, and finally to its standard lifecycle and interaction with the system. The gray boxes in Figure 6.4 are the major components that have been modified to support AFrame. In this section, we drill down to these components, and describe the design and implementation of AFrame to achieve the four isolation goals.

6.2.1 Process Isolation

The objective of process isolation is to run the AFrame activity in a different process with a different user ID, so the AFrame activity is physically separated from the application's main activity. Compared to the original Android code, we need to create a new user, a new process and a new activity for the AFrame region.

Application Installation and Setup. The information about AFrame needs to be retrieved by Android when an application is installed. During installation, Android parses the application's manifest file, and retrieves the application's component information, including activities, services, broadcast receivers, and content providers. Android also creates a new user for this new application, as well as creating a private data folder for its resource storage. This is done by the Package Manager Service (PMS).

In the design, a new parsing module has been added to PMS, which parses the `<aframe>` tag defined in the manifest file to retrieve AFrame information, including its activity class name and the permissions assigned to the AFrame. Based on the information, the modified PMS will create an additional user for the AFrame, and set up the private

data folder. The procedure conducted for the AFrame is exactly the same as that for the application.

Process Creation. When an application is launched, a process needs to be created to run the application. In Android, this process creation is initiated by the Activity Manager Service (AMS). To start a process, AMS first requests the process information from PMS, including the process's user ID (UID), group ID (GID), the groups that the UID belongs to (GIDs), and the data folder (see the left part of Figure 6.4). After getting the information, AMS sends a process-creation request to the Zygote process, which forks a new process, and configures the new process using the process information.

To support AFrame, the above procedure has been slightly modified . In addition to creating a new process for the application, AMS also retrieves the AFrame process information from PMS, if the application contains an AFrame. It then sends an additional request to Zygote, requesting it to create a new process for the AFrame. The main process and AFrame process are started simultaneously.

Activity Loading. After the process is created, Android needs to load the corresponding activity into the process, and bootstrap the activity's lifecycle, so the application can interact with users and the system. In the AFrame design, in addition to loading the application's activity to its process (already done by Android), we need to load the AFrame activity as well. Process record, activity record and runtime context are three essential resources for activity loading and for activities to run properly. After all these

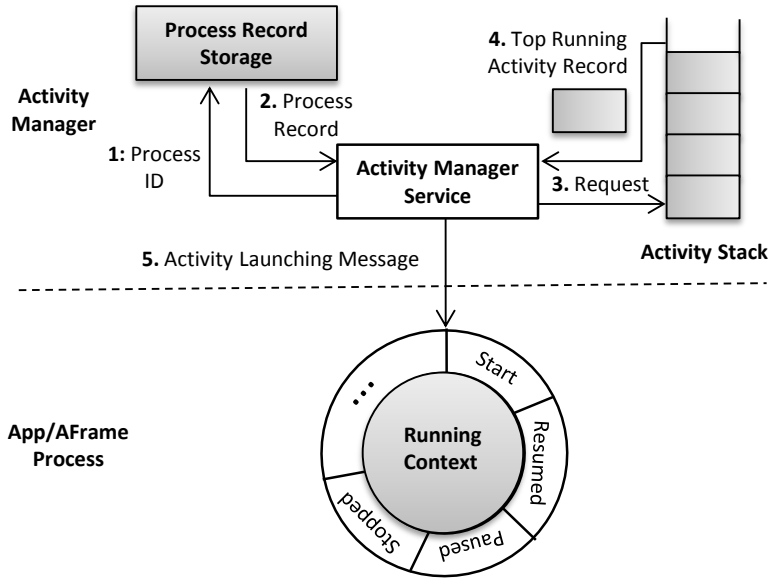


Fig. 6.5.: Android Activity Loading

types of resources are prepared, Android loads the activity under the control of AMS, following the steps depicted in Figure 6.5.

The current activity implementation has been inherited as much as possible to construct the above process record, activity record, and runtime context for the AFrame activity. Following this principle, all the values of the main activity are copied first, followed by the manual identification of all necessary places where the AFrame activity should have different values. For example, in the AFrame process record, the process name and ID need to be changed. Since the AFrame component still belongs to the application, there is no need to change its application information. Similarly, for the AFrame activity record, its process name, data folder and hosting process record have been changed. However, for the state of the current activity, we can keep the value since the AFrame activity always shares the same state with the main activity. Moreover, we need to create and register new communication with system services. For example, AMS needs to

communicate with the AFrame activity to control the transition of its states; Window Manager Service (WMS) needs to communicate with it to set the visibility of its display.

Once the activity record is constructed, AMS constructs an activity launching message with the activity record as the argument and delivers it to the hosting process using the activity token. The UI thread in the hosting process responds to the request by initializing the context of the current activity, loading the activity class and triggering its lifecycle. For AFrame, another activity launching message has been construed based on the AFrame activity record, and delivered to the AFrame process.

6.2.2 Permission Isolation

In Android, each application is assigned a unique user ID (UID) at the installation time, and each UID is associated with a set of permissions. At the runtime, Android uses the UID to find out the permissions of an application, and conduct access control based on the permissions. Since the AFrame activity's UID is different from the main activity's UID, their permissions are naturally isolated from each other. There is no need to change the access control logic of Android to support AFrame. Only 10 lines of code has been changed for permission isolation, and none of them affects the access control logic. At the time the application is installed, a combination of the requested permissions from main process and AFrame process will be prompted to the user for consent.

6.2.3 Display Isolation

We describe how the AFrame activity and the main activity can share the same screen, while not being able to tamper with each other's display region. Before that, it is essential to explain how drawing works in Android.

(A) How Drawing Works in Android

Graphic Buffer Allocation. The drawing memory required by applications is managed by a system process called *SurfaceFlinger*. To be able to draw, an application first needs to request a buffer from SurfaceFlinger that interacts with the hardware abstraction layer (HAL).

Android has two types of HAL: gralloc is used for emulator and framebuffer is used for real devices. For emulator, it uses double buffering, so the HAL allocates a screen-sized graphic buffer. This acts as an off screen image and is called back buffer. The back buffer is used for drawing. When the drawing completes, this buffer is copied using block line transfer to the screen surface, called primary surface. Double buffering is used to eliminate visible draws.

On real devices, page flipping is used to eliminate tearing. In this case, the HAL uses twice the screen size to allocate memory. The purpose is to use one as a back buffer and the other as the primary surface. The drawing is done on the back buffer, while the contents of the primary surface are used to display current screen contents. When the drawing is complete on the back buffer, page flip is conducted, so the back buffer becomes the primary surface, and the old primary surface is now used as the back buffer.

For both types of HAL, hardware graphics operations need physical continuous memory to manipulate. SurfaceFlinger pre-allocates a fix-sized (8M, by default) chunk of memory using Process MEMory allocator (PMEM). PMEM is typically used to manage large (1-16+MB) physically contiguous regions of memory shared between userspace and kernel drivers. Upon graphic buffer requests from the application, SurfaceFlinger chooses the next available block of memory and sends the specification details back. The application takes advantage of the file descriptor, base address, offset and block size information inside the specification, and maps the same block of memory to its own process. The application process further packages the mapped buffer into a Java object, called *Canvas*, and provides drawing APIs to applications through this Java object.

Display Rendering. After the activity finishes its drawing, it informs SurfaceFlinger to do the rendering. This time, SurfaceFlinger functions like a window compositor and can combine 2D/3D surfaces from multiple applications. In Android, each activity window corresponds to a layer, and the layers are sorted in Z-order. SurfaceFlinger renders the display in two phases:

- *Visible Region Calculation:* From top to bottom, each layer takes its height and width as the initial visible region, and then subtract the region covered by the layers on top of itself.

- *Layer Composition:* From bottom to top, SurfaceFlinger copies the data inside each layer's visible region to the main surface. Main surface is a frame buffer specific for holding the layer composition result.

As Android commonly uses the standard Linux frame buffer device, HAL uses the frame buffer to generate the final image, which is sent to the frame buffer driver in the Linux kernel for displaying.

(B) Display Isolation

The AFrame activity and the main activity must share the same screen, and at the same time, their drawings should be restricted to their own regions. The combination of sharing and isolation makes the design for this part very challenging. two design choices have been considered: achieve sharing in graphic buffer allocation but isolation in canvas drawing (soft isolation), or achieve isolation in graphic buffer allocation but sharing in display rendering (hard isolation).

Soft Isolation. In this design, after SurfaceFlinger prepares a graphic buffer, it maps the same buffer memory to both the main process and the AFrame process. Essentially, both activity processes share the same buffer, and can freely draw anywhere on the screen. We need to restrict their drawings to their own regions. There are two ways to access the graphic buffer. One is through the direct memory access using native code. If the activity inside AFrame does not have its own native code or the native code brought by it is blocked, this path is blocked. The other is through the standard Canvas APIs to draw some objects in the buffer. These APIs implement a clipping mechanism to ensure that the drawing by each node on the view tree can only affect the region assigned to that node, and nothing beyond. Therefore, all we need to do is to set up the clipping region correctly

for the main activity and the AFrame activity, so their drawings using the Canvas APIs are always restricted to their own designated regions.

The lightweight soft isolation comes with a cost of security, since it depends on the unlikely assumption that there is no native code execution in android advertisements. To make AFrame more secure, the following hard isolation option has been adopted in the implementation.

Hard Isolation. In this design, the main process and the AFrame process do not share a single drawing buffer any more. Instead, each process gets a unique graphic buffer from system, and maps that block of memory to its own process space for drawing. As a result of that, the memory space of these two processes are totally isolated and accessing the other process's drawing memory will result in a hardware exception.

After each activity finishes its drawing, two layers, i.e., main layer and AFrame layer, are used by SurfaceFlinger for display rendering. In order for SurfaceFlinger to distinguish the main layer from other layers, we send its `AframeReserve` region information to SurfaceFlinger before the graphic buffer allocation request. During the *Visible Region Calculation* phase, we identify a layer as the main layer if SurfaceFlinger contains its `AframeReserve` information. In such case, the `AframeReserve` region is cropped from its visible region. As a result of that, the AFrame layer below is able to keep the `AframeReserve` region as its visible region following the original *Visible Region Calculation* phase.

When SurfaceFlinger finishes the layer composition, the final image will be a combination of the drawing from main activity and AFrame activity, each activity only

contributing to its designated region. Since the composition happens in SurfaceFlinger process, which is a privileged system process, there is no way for applications to tamper the designated logic, even with the native code considered.

6.2.4 Input Isolation

Events should be considered as application's private resources, because they affect application behavior. Normally, events are generated by user interactions, such as clicking, touching, and key strokes, but Android also allows applications to generate events programmatically. While events injection to an application itself is always allowed, injecting events to another application needs the *INJECT_EVENTS* permission, which is a system-level permission that is never granted to normal applications.

How Event Dispatching Works in Android. When a new activity is started, it sends a request to the WindowManager system service to register an input channel with the system. This channel will be used by the system to send events to the activity process. WindowManager forwards the request to InputManager, which responds to the request and establishes the input channel with the new activity. WindowManager manages the z-order of activity windows, and synchronizes this information with InputManager. This way, they both know which activity is on the top and is currently receiving focus from user interaction. When InputManager receives an event from the device driver or applications, after checking the *INJECT_EVENTS* permission, it delivers the event to the activity on the top through the established input channel with the activity. Once the activity receives

the event, it further dispatches the event down its view tree until some view object consumes the event. The event will be discarded if no one can handle it.

Input Isolation. The original Android system assumes that only one activity is on the top and receives input focus. With AFrame, this is not true anymore, so we need to make corresponding changes to the system. The original input-channel registration process has been extended, so InputManager also knows which display region belongs to the main activity or the AFrame activity. An additional *Decision Maker* module has been added to InputManager to enforce the isolation.

When InputManager receives an input event from the system, it first decides which window is on the top, and then decides which activity (main or AFrame) should get the input. Because InputManager knows the region information and the event location, it can choose the correct input channel and dispatch the event to the intended activity process.

The above change is not sufficient. In Android, an activity can generate a user event; this event will be sent to InputManager, which will identify the target input channel, check the *INJECT_EVENTS* permission and then dispatch it. The input channel re-choosing mechanism may change the target channel, and therefore, violate the input isolation. For example, the main activity can inject an event to itself but target the AFrame activity region. This event injection will be allowed since it targets at its own process. However, the input channel re-choosing mechanism redirects the event to another input channel, which is in the AFrame process, resulting in a violation of input isolation.

To achieve input isolation, an additional access control module has been added in InputManager. Before the events are actually dispatched to the targeted input channel, the

module checks again whether the UID of the sender process is the same as the UID of the target process. If these two UIDs are the same, event dispatching is allowed, i.e., an activity can generate an event for itself. However, if the two UIDs are not the same, the event will not be dispatched, unless the event generator has the system-level *INJECT_EVENTS* permission.

6.2.5 LifeCycle Synchronization

AFrame activity and main activity are running in different processes, but they have to function like one activity; namely, they have to "stick" together in terms of visibility, behavior and existence.

The activity lifecycle begins with instantiation, ends with destruction, and includes many states in between. All the state transitions of activities within an application are managed by AMS. When an activity changes its state, the appropriate lifecycle event method is called, notifying the activity of the impending state change and allowing it to execute developer's code in order to adapt to that change. After the activity finishes its transition, it notifies the AMS, so the system can react to the activity's new state.

Synchronization between an AFrame activity and its host activity requires additional communication between AMS and the application. Whenever AMS receives the notification from the main activity, it also notifies the AFrame activity of the same impending state change, so the AFrame activity can go to the same state transition.

6.2.6 Code Modification

AFrame is implemented in Ice Cream Sandwich (ics): *android-4.0.3_r1.2*. Table 6.1 measures how many changes made to the original Android source code. Totally, 45 new functions are added and 23 existing ones are modified. The modification only introduces around 1200 new Lines Of Code (LOC), including 700 lines of Java code and 500 lines of C/C++ code. A small change has also been made in the XML file to add the `aframe` tag in the Manifest file. .

Important Components	Modules	# Functions		# LOC	
		Modified	New	C/C++	JAVA
AFrame Parsing	AFrame Parsing	2	2	0	80
Process Isolation	AFrame Info Retrieve	1	5	0	30
	Process Creation	1	0	0	50
	Activity Loading	1	0	0	30
Permission Isolation	AFrame Permissions Retrieve	0	2	0	10
	Permission Enforcement	0	1	0	10
Display Isolation	Graphic Buffer Allocation	10	20	500	50
	Restricted Canvas Usage	1	2	0	70
Input Isolation	Effective Canvas Info Retrieve	0	12	50	20
	Decision Maker	0	1	150	0
	Restricted Event Injection	1	0	10	0
Sync	Synchronization	6	0	0	150
Others	AFrame Tag Declaration	About 5 new lines in XML file			
	AFrame ViewGroup	New class: 10 lines of Java Code			
Total		23	45	710	500

Table 6.1: Lines of Code (#LOC) Added to AOSP Android for AFrame

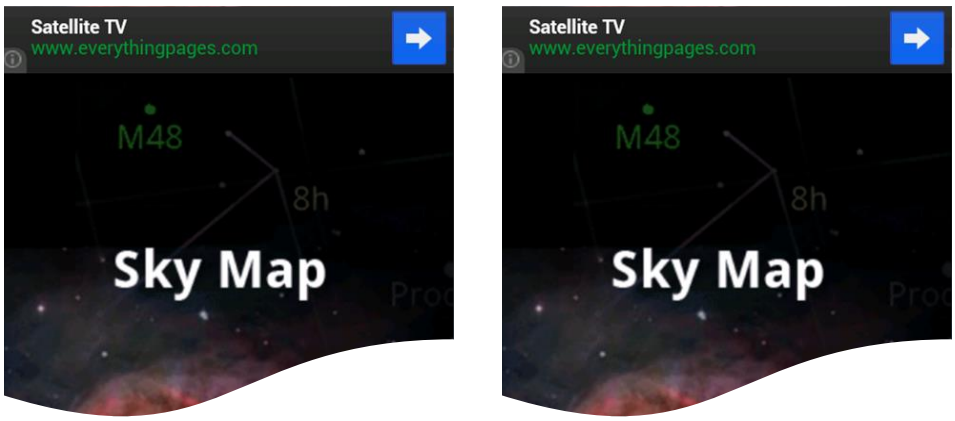
6.3 Evaluation and Case Studies

The evaluation of AFrame focuses on two aspects: effectiveness and performance. For effectiveness, advertisement is used as a case study to demonstrate how AFrame can be effectively used to isolate the privilege of untrusted components from that of the main app. For performance, the system-level and application-level overheads caused by AFrame have been measured.

6.3.1 Isolating Third-Party Advertisements

This experiment shows how AFrame can protect apps against untrusted 3rd-party advertisements. Because of the need to modify the existing apps, an open-source app, Sky Map [110], is used as the hosting app. According to the statistics on Google Play, this very popular app has been downloaded more than 10 million times. Donated by Google, Sky Map does not include any advertisement. For the demonstration purpose, its source code is manually modified to incorporate the AdMob advertising SDK. Figure 6.6 shows what the app looks like. From the figure, we can see that there is no visual difference between the one using AFrame and the one without AFrame, i.e., AFrame is transparent to users.

Privilege Isolation. Sky Map requires the following permissions: *INTERNET*, *WRITE_EXTERNAL_STORAGE*, *READ_EXTERNAL_STORAGE*, *READ_PHONE_STATE*, *ACCESS_FINE_LOCATION*, *ACCESS_NETWORK_STATE*, *WRITE_SETTING*, and *WAKE_LOCK*. Several of these permissions are related to user's private data. Once granted, they can be used by any



(a) With AFrame (b) Without AFrame

Fig. 6.6.: Sky Map with Advertisement

component of the app, including the advertisement. Therefore, the 3rd-party advertisement can collect the user's location data, phone state, and read the data stored on the SDCard.

With AFrame, we can easily limit the privilege of the advertising component, without affecting the original app. To achieve that, in the Sky Map's manifest file, the advertisement activity is declared in an `aframe` tag. Only two permissions—required by AdMob—are declared for this AFrame region, including *ACCESS_NETWORK_STATE* and *INTERNET*. The detailed specification is shown below:

```
<aframe android:name=".activities.AdsActivity">
  <aframe-permission
      android:name="android.permission.INTERNET"/>
  <aframe-permission
      android:name="android.permission.ACCESS_NETWORK_STATE"/>
</aframe>
```

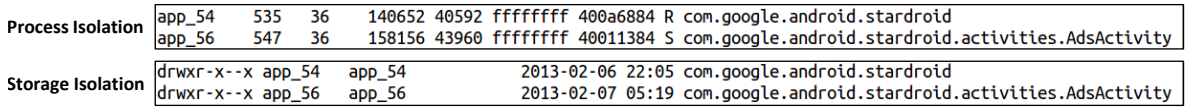


Fig. 6.7.: Process and folder isolation between main activity and AFrame activity with different UIDs

As Figure 6.7 shows, when the advertisement activity is started, it is loaded into a different process with a different user ID (see Figure 6.7). Moreover, the AFrame activity and the main activity have their own data folders, and the permissions on these folders ensure that one activity cannot access the other activity's data.

To further demonstrate the privilege restriction on the advertisement process, a piece of malicious code is intentionally embedded in the advertisement activity and tries to access the user's location. As Sky Map runs, a security exception is thrown from the advertisement process (Figure 6.8), indicating that the developer has not declared the *ACCESS_FINE_LOCATION* or *ACCESS_COARSE_LOCATION* permission. Without using AFrame, this access will be successful, as the Sky Map app has these permissions.

Since the above exception has been handled in the advertisement activity code, the Sky Map app does not crash and can continue to run smoothly. If the exception handling code is removed, the AFrame process will crash. Because AFrame runs in a different process, the main process does not need to terminate when the AFrame process crashes. Whether the app should be required to terminate or not depends on the synchronization policy set by the developer. It should be noted that without using AFrame, if the advertisement crashes, the entire app will crash.

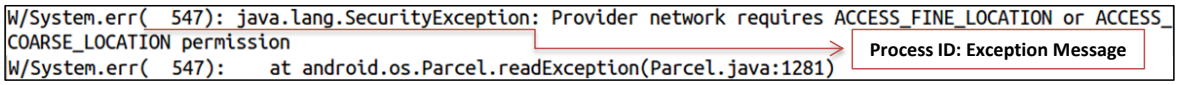


Fig. 6.8.: Security exception caused by not having the location permissions in the AFrame process

Compatibility with various advertising libraries. The above experiment has been repeated with other advertising libraries. Five popular advertising libraries are selected based on the statistics provided by the 3rd-party Android market AppBrain [111]. They all work with AFrame, and the user experiences are similar to that of AdMob. Table 6.2 summarizes the results.

Other Applications. AFrame has been further tested with other open-source apps, including Easy Random Numbers, Easy Graphic Paper and Skillful Lines [112]. All of them were released in 2012, with the AdMob library already added. Each of the selected apps was modified to move the AdMob code into AFrame. This involves adding a new activity for AFrame, reserving space for AFrame in the original activity layout file, moving the layout for advertisement to the AFrame layout file, and specifying permissions for the AFrame in the manifest file. This modification is quite simple, and takes about 10 minutes for each activity. All apps run successfully on Android, and the observations are similar to those obtained from the Sky Map experiment.

6.3.2 Performance: System Overhead

To evaluate the performance of AFrame, the following environment setup has been used: The source code is based on Ice Cream Sandwich(ics): *android-4.0.3_r1.2*, and the

	% of apps	% of installs	Compatibility
AdMob	32.34	44.17	Yes
Millenial Media	3.73	16.67	Yes
TapJoy	1.24	11.85	Yes
InMobi	2.62	12.54	Yes
Greystripe	0.39	1.38	Yes
Mopub	0.71	5.15	Yes

Table 6.2: AFrame Compatibility with Ad Libraries

AuTuTu	Original SDK	AFrame SDK	Difference
Total Score	7302.90	7250.00	-0.71%
Memory Access	1413.15	1410.35	-0.20%
Integer Operation	1697.30	1689.90	-0.44%
Float Operation	1292.25	1286.80	-0.42%
2-D Graphic	460.20	451.80	-1.83%
3-D Graphic	1860.25	1833.15	-1.46%
Database Operation	312.00	311.75	-0.08%

Table 6.3: Benchmark Scores

modified Android runs on Samsung Galaxy Nexus I9250 with the following specifications: CPU (Dual-core 1.2 GHz Cortex-A9), RAM (1 GB), Display (4.65, 720*1280), GPU (PowerVR SGX540), Internal Storage (16 GB), and no SDCard.

To measure the overhead imposed on the system, a popular Android benchmarking tool, AnTuTu [113], is used. It runs a series of tests and provides a score report on various metrics, including performance on memory access, integer & floating-point operations, graphics, database I/O, and SDCard access (read/write). The higher the score is, the better. The results (average scores of multiple runs) are provided in Table 6.3. Since the testing phone does not have an SDCard and AFrame does not have any modification that affects SDCard, we remove the SDCard score from the final results.

From the AnTuTu benchmark, it can be observed that AFrame imposes no significant overhead on memory accesses, CPU, and database. It does impose a slight overhead on 2D

graphics (1.83%) and 3D graphics (1.46%). AFrame imposes these overheads because it performs additional checks before a canvas is created for drawing and AFrame also performs checks for layer composition before the final image is sent to the frame buffer driver in Linux kernel for displaying.

6.3.3 Performance: Application Overhead

We also need to evaluate the performance impact of AFrame on apps. The following aspects have been measured: memory consumption, time to start the app, and event dispatching. To conduct the tests, six testing apps are developed, which are divided into three groups. Each group has two apps with identical configuration, except that one uses AFrame and the other does not. The detailed specifications are shown in Table 6.4. Apps in Group A have a WebView component that loads a blank page, while apps in Group B have a similar configuration, but load the Google page. In contrast, apps in Group C have an advertising component that uses AdMob. As apps in Group C do not have a WebView component, the webpage configuration is not available for them.

Groups	In-Group Difference	AdMob	WebView	Webpage
A	No AFrame	No	Yes	Blank
	With AFrame	No	Yes	Blank
B	No AFrame	No	Yes	Google
	With AFrame	No	Yes	Google
C	No AFrame	Yes	No	N/A
	With AFrame	Yes	No	N/A

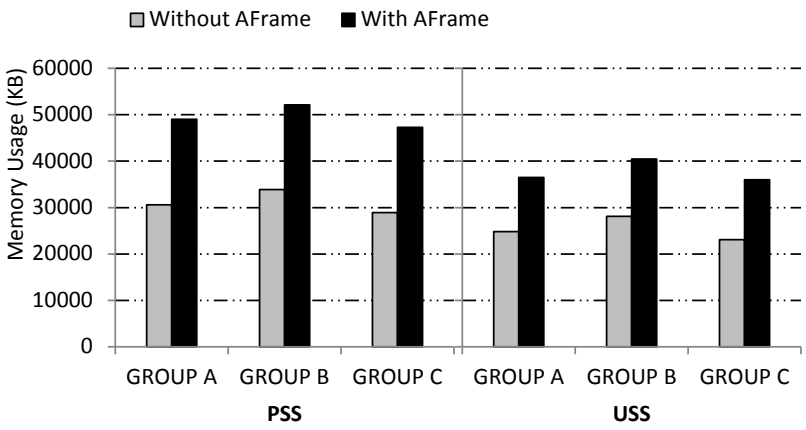
Table 6.4: Sample Applications for Overhead Evaluation

Memory Overhead. Procrank [114] is used to measure the memory overhead. This tool measures the memory usage by an app in terms of PSS (Proportional Set Size) and USS (Unique Set Size). PSS divides the size of shared memory equally among all the processes that share it, and count each process's contribution to the overall memory. The other measure, USS, only counts the amount of memory used uniquely by a process. For apps containing AFrame, the memory usage is the sum of the usage of the app process and the AFrame process. At first, each app runs individually and the measurements are depicted in Figure 6.9(a).

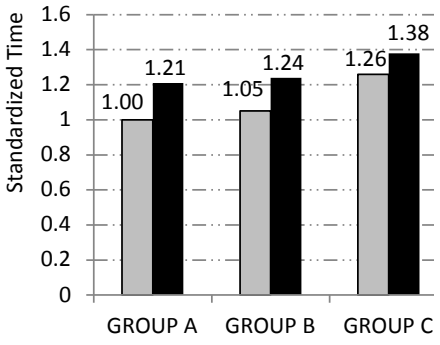
	Group Index	Pss Overhead	Uss Overhead
Individual Run	A	60.00%	47.00%
	B	53.00%	43.00%
	C	63.00%	56.00%
Average		**59.00%**	**49.00%**
Group Run	A	27.00%	43.00%
	B	25.00%	46.00%
	C	30.00%	47.00%
Average		**27.00%**	**45.00%**

Table 6.5: Memory Overhead Comparison

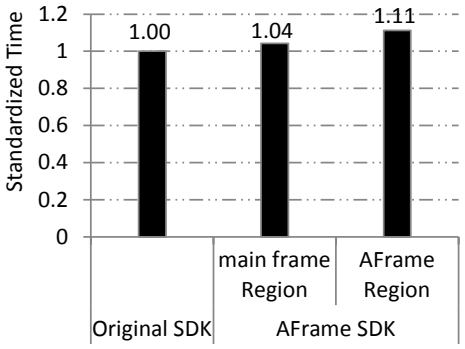
We do expect AFrame to increase the memory consumption quite significantly, as it adds an additional process to each activity. This is confirmed by the results in Table 6.5, as AFrame adds about 59% and 49% memory overhead to PSS and VSS, respectively. However, in real world, it is quite common that multiple apps are alive at the same time, with most of them running in the background. To better emulate our daily smartphone usage, we simultaneously run the AFrame app from each group in background and the normal app from the same group in front. Due to the memory sharing mechanism on android, the memory overhead to PSS dramatically drops to 27%, which is only less than



(a) Memory Overhead (Individual Run Case)



(b) Time-to-start Overhead



(c) Event Dispatching Overhead

Fig. 6.9.: Various Overhead Introduced by AFrame

half of the previous overhead. Even though VSS does not count the shared memory, the corresponding overhead still decreases slightly to 45%.

It should be mentioned that the current AFrame implementation have not exploited any optimization in the current implementation. We believe that the memory overhead can be further reduced using optimization. For example, if an app has N activities, each running an advertisement in its AFrame, instead of creating N additional processes, we can host all these advertisements in a single AFrame process. We can also reduce the heap size of the AFrame process, etc.

TTS Overhead. When an app uses AFrame, the app startup procedure involves checking the presence of AFrame component and starting an additional process. AFrame's effect on app's startup time (TTS stands for "Time To Start") has been evaluated. At first, the system time is recorded when launching an app in ActivityManagerService.java, before the process fork arguments are sent to Zygote via Process.start(). Then, the startup ending time is recorded in reportActivityLaunchedLocked() from ActivityStack.java. The startup time for the six sample apps are shown in Figure 6.9(b). The results are standardized based on the startup time of the app with WebView loading a blank page. As it shows, if AFrame is used to isolate AdMob library, the TTS overhead is only 9%. However, if AFrame is used for other purposes, such as isolating WebView, the TTS overhead will be a little bit higher. In the WebView case, the overhead will reach 20% (18% for loading blank page and 21% for loading Google), but the absolute time is still acceptable, because it is similar to the time for loading a normal advertisement.

Event Dispatching Overhead. When an app uses an AFrame component, when dispatching events to the app, the system will conduct additional checks to ensure that events for the main frame are not sent to AFrame, and vice versa. The overhead of this checking has been measured by calculating the difference in time taken to dispatch a single event. Two sample apps have been developed for this purpose. The first app contains an activity with a single button, and it is executed in the unmodified Android system. The second app contains an activity with two buttons, one in the main activity, and the other in AFrame. This app is executed in the modified system. The evaluation measures how long it takes a touch event to be dispatched to its corresponding button. Figure 6.9(c)

depicts the evaluation results, which show that the overhead for the main frame is quite small, and the overhead for the AFrame is about 11%. If we use AFrame to display advertisement, this overhead does not have much effect, as users do not interact with the advertisement very often [108, 109].

6.4 Conclusion

To totally isolate untrusted 3rd-party libraries, such as advertisement, from their hosting app, this chapter presents the design and implementation details of AFrame, a developer friendly framework to achieve the necessary process isolation, permission isolation, and input/output isolation. With AFrame, untrusted 3rd-party libraries can be isolated into a different process with a different UID. Moreover, developers still have the capability to configure the permissions they request. AFrame also ensures the integrity of each other's display and input resources. Several case studies have been conducted on advertising libraries and WebView components to demonstrate AFrame's effectiveness. The performance has also been measured to show that the overhead is acceptable. The results indicate that AFrame is a viable solution to solve the over-privileged problem associated with 3rd-party libraries.

7. SUMMARY

This dissertation has evaluated the security design of three program integration schemes on Android, including framework module, framework proxy and 3rd-party code embedding. After a systematic risk analysis, we have excluded the ones that have been addressed in Android's security architecture or extensively studied in existing research literature.

Throughout the process, we have uncovered the new data residue vulnerability. In particular, after an app is uninstalled, the Android framework still keeps sensitive information belonging to that app, leading to data residue instances. The vulnerability arises if the protection on those data residue instances becomes ineffective in preventing unauthorized access from adversaries. In this dissertation, we have manually inspected the source code of one particular Android version to demonstrate the severity of the data residue vulnerability. Then, we have designed and implemented ANRED, an Android residue detector, to automatically detect data residue instances from vendor images. Our study has provided the first comprehensive understanding of the data residue situation across the entire Android ecosystem.

In addition, we have designed and implemented AFrame for Android to separate the privilege of untrusted advertising libraries from host apps. AFrame supports a similar "frame" in Android activities as iframe in web pages. From the user perspective, these two activities look like one because they seamlessly appear on the same window and behave like one unit. However, from the system perspective, they actually run in two different

processes with different user IDs. AFrame achieves process/permission isolation that is also in existing works, but it goes beyond that by providing display and input isolation. Another important advantage of AFrame is that using AFrame, developers can use the advertising SDK like before, all they need to do is to tell Android to load the advertising code in a special region.

In this dissertation, we have provided insights for Google to improve the security design of program integration schemes on Android, and urged device vendors and app developers to take more responsibility in securing their Android products. Future works can be focused on four aspects.

Eliminating Android Data Residues. Our studies in Chapter 4 and Chapter 5 present an accurate and comprehensive understanding of the data residue situation across the entire Android ecosystem. However, additional efforts are required from the research committee to eliminate the data residue vulnerability completely. First of all, it is imperative to provide design guidance for system services that would explicitly address the data residue problem. The design should clearly specify whether there is a potential data residue, whether apps' data are removed when their owner apps are uninstalled, and if not, what security consequences might occur if the data are inherited by other apps. Second, it is necessary to systematically evaluate all attributes used in Android to enforce access control policies. Android implicitly assumes that these attributes are unique to individual apps. However, our study has demonstrated that uninstallation and device reboot can invalidate the assumptions, leading to re-association of some attributes to a different app. Another research direction is to effectively taint all data saved in Android system services

with respect to their owning entities. Thus, we can detect and remove data residue instances when they emerge. The challenges lie in propagating taint labels accurately and covering all conditions that can trigger data residue instances.

Data Residue on Other Platforms. Android is not the only platform that has the data residue concern. In traditional computing platforms, when an app is uninstalled, its data still belong to the users, and the security parameters of those data do not change. Thus, the data residue vulnerability has larger implications on platforms that treat each app as a different user. Examples include other mobile OSes and the extension framework supported by most mainstream browsers. In particular, each extension on Firefox can be considered as a single app, which follows the installation, interaction and uninstallation lifecycle. A similar research question is that, after a Firefox extension is removed, whether there will be data residue instances, and if so, whether Firefox's security architecture prevents unauthorized access from adversaries.

Cross-frame Communication Support in AFrame. The AFrame work presented in Chapter 6 mitigates the threat of over-privileged advertisements in Android apps. The current implementation of AFrame allows developers to isolate advertising libraries into different processes. However, the communication between advertising code and their host apps is not supported. There are two improvements that can be done in future research. First, multiple AFrame instances can run in the same process to reduce memory overhead. The selection of AFrame instances should take into consideration their host apps, the source of advertising libraries and permission requirements. Second, additional research

efforts are expected to support cross-frame communication in AFrame. In this process, the research challenge is to identify the appropriate set of APIs that should be allowed during the interaction.

Selecting Isolation Unit. Although the prototype of AFrame is only available on Android, the design experience can be beneficial to other platforms. One important lesson learned is to carefully select the basic isolation unit when designing future operating systems. In Android, isolation is provided at the process level. Thus, it does not address the security risks at the component level, resulting in over-privileged advertisements. In contrast, browsers treat the origin of web pages as the smallest isolation unit, which enables the separation of advertisements embedded in iframe from their host web pages. The selection of isolation unit should take into consideration all potential risks faced by the operating system. In addition, it is also essential to employ proper techniques in achieving the designated isolation. Originally, browsers use sandboxing technique to separate the security context of iframes from their parent document. However, as they are still running in the same process, the speed, stability and security are of major concerns to end users. To support out-of-process iframes (OOPIFs), Chromium has done a massive refactoring effort. Similar engineering efforts have also been observed in AFrame's implementation to support display sharing among multiple processes. An interesting research question is how to properly select the isolation unit and enforcement techniques at the initial design of operating systems in order to avoid massive refactoring effort in the future.

A. ATTACKS ON ANDROID CLIPBOARD

Besides the security analysis of program integrations in Android, the security risk in Android Clipboard has also been studied. To be more specific, everything placed on the clipboard is public and accessible to all the running apps on the device without any permission requirements or user interaction. Android even allows apps to monitor data changes on the clipboard by registering a callback listener to the system. This is not a severe security problem on the desktop environment, since its clipboard is user-driven and a window should transfer data to or from the clipboard only in response to a command from the user [115]. But still, attacks caused by the clipboard on desktop environment have been observed in past few years. For example, Self-XSS attack [116] happens when an attacker convinces a user to copy-paste some malicious JavaScript code in his/her URL bar and hit 'Enter'. Most of these attacks involve significant social engineering efforts from attackers to trick victim users to conduct desired operations. Two solutions [117, 118] have been proposed and implemented to fix the Self-XSS vulnerability. While Firefox disallows the execution of JavaScript in its URL bar, Google Chrome takes a less restrictive approach by removing the "javascript" tag when copying and pasting a URL. However, it has been demonstrated that attackers are still able to bypass the protection on Google Chrome by placing a control character just before the "javascript" string [119].

In contrast, Android considers each app as a different user with different privileges. Due to the global unguarded access, various users, i.e., apps, can arbitrarily operate on Android

Clipboard without any restriction. What makes the situation worse is the limited screen size of mobile devices. First of all, users are much more likely to copy and paste data to save typing efforts, opening up plenty of attacking opportunities. Furthermore, fewer characters will be visible to users after pasting the content from the clipboard to the app, easing attackers' efforts in hiding their malicious behaviors. Another advantage for attackers targeting Android Clipboard is the lack of security awareness in common app development, even among top developers.

To understand the current security situation on Android Clipboard, this chapter presents the first systematic study of the clipboard usage in benign apps and malicious apps. The malware sample [120] consists of 3,987 malware apps collected from different sources [121]. The benign sample consists of the top 500 free apps in each category in Google Play (around 16,000 apps), and they were collected in July 2012. The analysis result shows that 1180 benign apps provide the functionality to put data on the clipboard, while 8 malware apps try to retrieve data from the clipboard. Due to the open access, those 8 malware apps can easily steal whatever information leaked from the mentioned benign apps. At the same time, 384 benign apps are found to be able to get data from the clipboard. However, around 60 malware apps are capable of manipulating the data on the clipboard. If a benign app takes the clipboard data for execution without proper checking, any one of the 60 malware apps could possibly launch the code injection attacks.

Based on the risk assessment, two groups of attacks have been formulated, i.e., manipulation and stealing. Clipboard data manipulation may lead to common code injection attacks, like JavaScript injection and command injection. Furthermore, it can also cause phishing attacks, including web phishing and app phishing. Data stealing happens

when sensitive data copied into the clipboard is retrieved by malicious applications. The exploit details are presented in the following sections.

A.1 Short Tutorial on Android Clipboard

On Android platform, the clipboard is a powerful framework to support various types of data copy and paste within an app as well as among apps. To copy certain type of data, a corresponding clip object (*ClipData*) is constructed and placed on the clipboard if the required permission is granted to the app. A ClipData contains a description (*ClipDescription*) of the important meta-data about the clip. At the same time, it also holds one or more item instances (*ClipItem*). The clipboard holds only one clip object at a time. When an app puts a clip object on the clipboard, the previous clip object is erased. To paste data, the app retrieves the clip object and selectively handles the resolved data based on its MIME type. Different from copying data to the clipboard, no permissions are required for an app to access the content from the Clipboard. Moreover, apps can even monitor primary clip changes by registering a listener callback.

ClipManager is responsible for managing the copying, monitoring and pasting operations on the clipboard. Applications can simply access the *ClipManager* without requiring any specific permission, as shown in the following example:

```
ClipboardManager mClipboard = (ClipboardManager)
       getSystemService(Context.CLIPBOARD_SERVICE);
```

Once the *ClipManager* interface is retrieved, Android apps can use it to interact with the clipboard in three different ways; the details will be discussed in the rest of this section.

A.1.1 Copying Data from Apps to the Clipboard

Android clipboard supports the copying of various data types, including plain text, html text, intent, raw URI and content URI. While a string of text can be directly put on the clipboard, intent and URI are always considered as identifiers of the real data storage. In order to put an intent or URI on the clipboard, the app must hold the required permission to access the corresponding data entries. For each data type, the *ClipData* class provides a unique API to facilitate its construction. As an example, the following code shows how to copy Contacts information to the clipboard:

```
// Contact URI and ClipData Construction
contactUri = ContactsContract.CommonDataKinds.Phone.CONTENT_URI;
ClipData contentUriClip =
       ClipData.newUri(getContentResolver(), "LABEL", contactUri);
// Put the new ClipData on clipboard
mClipboard.setPrimaryClip(contentUriClip);
```

A.1.2 Monitoring the Clipboard Data

Android apps can monitor content change on the clipboard via the *addPrimaryClipChangedListener()* hook provided by ClipboardManager class. Apps use

this API to register a listener callback, which is invoked when the primary clip on the clipboard changes. Once triggered, the app can respond to the new ClipData with its own implementation of *onPrimaryClipChanged()* function inside the registered *OnPrimaryClipChangedListener* object. The following code shows a concrete example on how to attach such a callback function to monitor the clipboard:

```
// Register the Listener
mClipboard.addPrimaryClipChangedListener(new
    ClipboardManager.OnPrimaryClipChangedListener(){
        public void onPrimaryClipChanged(){
        // Handle the content changing event here
        }
    }
);
```

A.1.3 Pasting Data from the Clipboard to Apps

When pasting from the clipboard, Android apps first retrieve the clip object using another API called *getPrimaryClip()* in the ClipManager class. After that, they can selectively handle the retrieved clip data based on its MIME type. To elaborate, the following example only handles the content URI with MIME type *vnd.android.cursor.dir/phone*:

```
if(mClipboard.hasPrimaryClip()){
    ClipData clip = mClipboard.getPrimaryClip();
```

```
    Uri uri = clip.getItemAt(0).getUri();
    ContentResolver cr = getContentResolver();
    String type = "vnd.android.cursor.dir/phone";
    if(uri != null && type.equals(cr.getType(uri))){
        // Handle the content provider data
    }
}
```

The example above first ensures that the clipboard contains a clip; if so, it retrieves the primary clip. If the item is a content URI with the desirable MIME type, the app creates a ContentResolver and then calls the appropriate content provider method to retrieve the data. Different from copying data to the clipboard, apps do not need to have the corresponding permission in order to access the content identified by the URI or perform the operation specified by the intent. Instead, Android will temporarily delegate the permission to the app within the execution of *getPrimaryClip()*.

A.2 Threat Models

The attacks discussed in this chapter are categorized into two models based on the operations performed by malicious applications on the clipboard data, i.e., manipulation and stealing. This section will give a high-level overview of these two models (depicted in Figure 1), leaving the attack details to later sections.

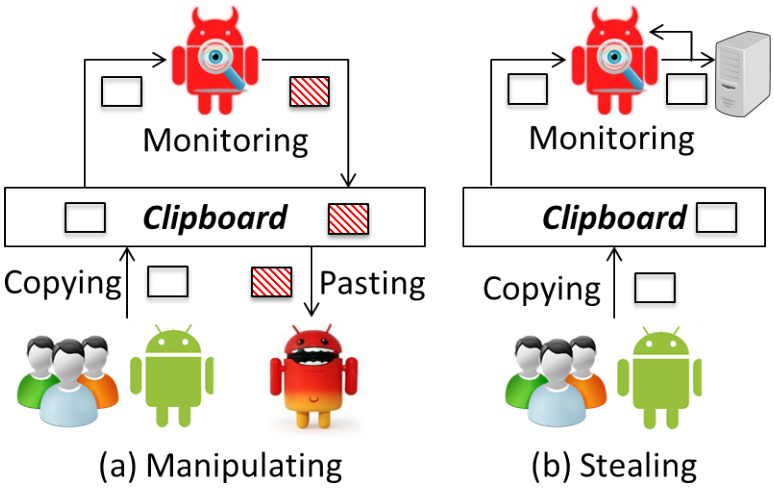


Fig. A.1.: Threat Models

Manipulation. Study has been conducted to show how malicious apps can interfere with other apps' execution by manipulating the data on the clipboard. This attack model assumes that the malicious app is installed on the same device as the victim app. The assumption is not very difficult to satisfy. Actually any app potentially can perform the attack, since it does not require any permission to access the clipboard on Android. The malicious app keeps monitoring the data change on the clipboard. Once the copying operation is performed either by some other benign apps or the user, the malicious app can selectively manipulate the data. When the modified data is pasted to the same or another app and that app's future behavior depends on the pasted data, the attack succeeds. For web-based apps, attacker can try to inject JavaScript to achieve various damages (Section A.3). For terminal apps, malicious commands may be injected to local/remote server for execution (Section A.4). The attacker can even perform phishing attacks on social websites as well as their applications (Section A.5).

Stealing. Study has been conducted to show how malicious apps can steal user's private information, which leads to data leakage attacks. The assumption for this threat model is the same as the previous one. However, instead of manipulating the data, the attacker tries to detect user's private data on the clipboard and steals it (Section A.6). The attack will cause more damage if the data on this clipboard is a URI or Intent, which serves as an identifier to user's private information, such as Contacts, Calendar or Messages. Although this may sound less likely to happen, the above requirement is not difficult to achieve at all. Firstly, it is not rare for users to copy their username or even password to the clipboard. Secondly, many apps available on Google Play allow users to perform private data copying and pasting, leaving plenty of attacking opportunities for malicious apps.

A.3 Injection Attacks - JavaScript

A.3.1 JavaScript on Mobile Browser's URL Bar

An emerging trend among all browsers is the combination of searching and navigating from the same box, referred to as *URL Bar* in this section. When users are attracted by something they see on the web, they can type, or more commonly, copy and paste it into the URL Bar to directly search more information about it.

Considering that Android Clipboard is globally accessible to all the apps on the same device without requiring any permission, a malicious app can modify the content on the clipboard and inject malicious JavaScript code with some small tricks to hide the attack from the user's attention. Figure A.2 illustrates the phases involved in such an attack.

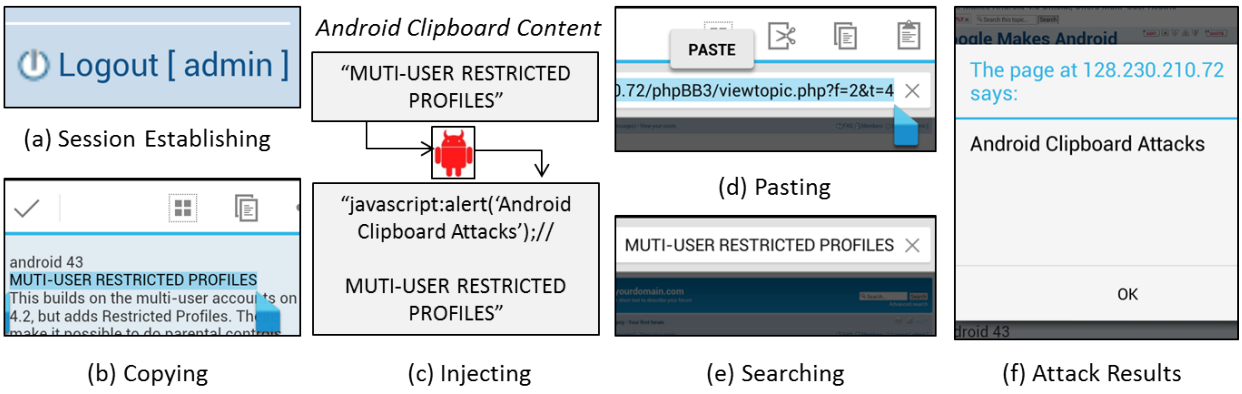


Fig. A.2.: JavaScript Injection on Vulnerable Browser's URL Bar via Copy-and-Paste

The success of the attack relies on the browser setting of JavaScript execution in URL Bar and the trick applied by attackers to hide themselves from the victims. To study the influence of such attacks, a systematic analysis has been conducted on the default setting of the built-in Android browser and other top 10 browsers on Google Play. The testing device is Samsung Galaxy Nexus running Android 4.3 (JELLY BEAN). After manually installing each browser app, the following JavaScript code is typed into its URL Bar:

```
javascript:alert('Android Clipboard Attacks');
```

If an alert window is displayed, the browser allows JavaScript execution in its URL Bar by default. The maximal characters visible on the URL Bar has also been measured for each browser. The study results are included in Table A.1. Different from desktop browsers that usually disallow pasting JavaScript code to URL Bar, all the studied mobile browsers allow such an operation. However, Firefox and UC Browser do not support JavaScript execution directly from the URL Bar, making themselves immune to such attacks. To hide the attack from users' attention, attackers can simply add enough blank spaces before the

Android Browser Apps	# of Installs	JavaScript Execution	Visible Chars
Built-in Browser	N/A	✓	<26
Firefox	>10,000,000	✗	<33
Dolphin	>10,000,000	✓	<20
ONE	>1,000,000	✓	<23
Opera Mini	>50,000,000	✓	<40
UC Browser	>10,000,000	✗	<29
Chrome	>100,000,000	✓	<33
Opera	>10,000,000	✓	<33
Dolphin Mini	>1,000,000	✓	<24
Maxthon	>1,000,000	✓	<25
Boat	>1,000,000	✓	<23

Table A.1: Analysis of the URL Bar in Top Android Browser Applications

malicious code. The number of blank spaces depends on the largest number of visible characters in each browser's URL Bar. The goal is to make the malicious code invisible to victim users unless they scroll down to check all the characters in the URL bar.

To launch the attack, the malicious app simply implements a service that defines a listener callback inside. The callback is invoked whenever the primary clip on the clipboard changes, allowing attackers to inject JavaScript code. The attacking types include but are not restricted to session hijacking, confused deputy, integrity compromise and privacy leakage. However, the damage is limited to the current domain because of the Same Origin Policy (SOP) [73]. To demonstrate each type of attack, the latest stable phpBB version (3.0.11) [122] was installed on a Dell OPTIPLEX 760 desktop running Ubuntu 12.04. Except for Firefox and UC Browser that do not allow JavaScript execution in their URL Bar, all the other browsers are vulnerable to the mentioned attacks. In the following sections, all the sample attacks are conducted in Google Chrome on the testing mobile device, unless otherwise specified.

Session Hijacking. The attacking steps follow exactly the same as in Figure A.2, with the malicious JavaScript sending the victim's cookies to the remote server. After that, the attacker can gain unauthorized access to the victim's entire account. It should be noted that the current stable phpBB version (3.0.11) has already implemented several mechanisms to prevent against session hijacking attacks, including HttpOnly cookie [123], session IP validation and browser validation. During the demonstration, those three protections have been turned off. However, the following *Confused Deputy* attack does not require the adjustments on the phpBB3 server, and still being able to achieve the same damage.

Confused Deputy. Since JavaScript execution in the URL Bar is under the same context of the current page, the attacker can send malicious requests from there to the remote server and valid cookie will be automatically appended by browser. It is impossible for the remote server to distinguish the malicious requests from benign ones, leading to the *Confused Deputy* attack. All the mentioned protection mechanisms in phpBB3 will be defeated as well since malicious requests are sent from exactly the same browser (defeating browser validation) on the same mobile device (defeating session IP validation) with all the valid cookie value appended (defeating HttpOnly cookie).

Integrity Compromise. In this scenario, the attacker can modify the value of any field on the current page in an unauthorized or undetected manner. Even though the correct value will recover after refreshing the page, data integrity has already been compromised since accuracy and consistency of data cannot be maintained and assured over its entire

life-cycle. Figure A.3 shows how attackers can advertise themselves on Google home page within the current interactive session on the victim user's mobile browser.

Fig. A.3.: Integrity Compromise on Google Website

Privacy Leakage. With the JavaScript injection attack on mobile browsers' URL Bar, attackers are able to steal sensitive information from victims, leading to *Privacy Leakage*. The most straightforward attack tries to steal the information of the browser itself, including type, version, resolution, history and bookmarks. Moreover, leveraging on the HTML5 technology, advanced attackers can also steal victim's GeoLocation information and everything stored in the local storage. Figure A.4 illustrates the possibility of privacy leakage from Facebook webpage. As it turns out, Facebook even locally stores telephone numbers of the victim's friends.

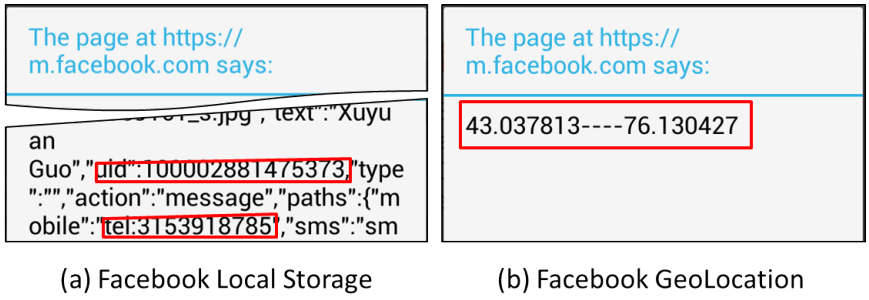


Fig. A.4.: Privacy Leakage on Facebook Application

A.3.2 Cross Site Scripting (XSS) Attack

Different from normal XSS attacks, the clipboard based XSS attack happens when the victim pastes malicious JavaScript code (manipulated by attackers) into a vulnerable app. As a result of that, the data pasted from the clipboard is reflecting the purpose of malicious attackers, while the operations are still conducted by the trusted device owner.

In the study, one vulnerable Android app[1], is found with more than 1,000,000 installs. The app itself is developed using standardized web APIs based on the PhoneGap [124] framework, and thus compatible with various mobile platforms, such as iOS, Android, Windows OS and etc. Unfortunately, its user profile form has XSS vulnerability. When the owner is creating or updating his/her profile, if the content is pasted from the clipboard, malicious apps can launch XSS attacks targeting at the victim app. The vulnerability detection techniques and potential damages of XSS attacks are well studied in previous work [125–129].

A.3.3 Cross Origin Invocation Attack

Both Android and iOS support the *scheme* [130, 131] mechanism, through which cross origin invocation becomes possible, i.e., an app (origin: application) can be invoked by a URL (origin: web) once it registers the URL's scheme. On Android, registration happens by simply declaring an intent filter in the app's manifest file. For example, activity with *android:scheme="fbconnect"* inside its intent filter can be launched by *fbconnect://...* typed of links.

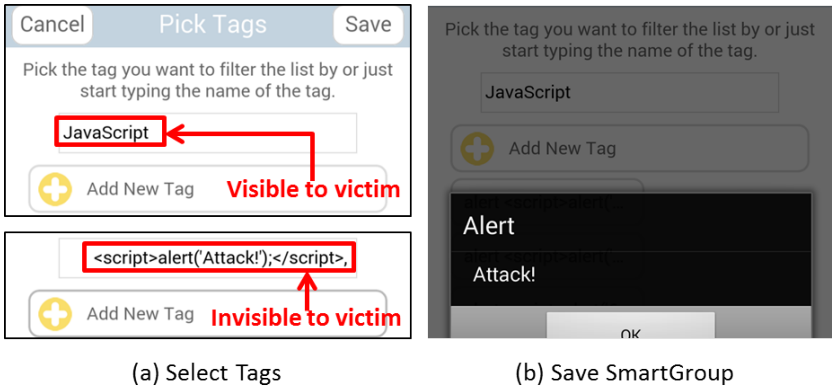
[1]To protect the company, its name is not disclosed.

Previous studies [36] have demonstrated the possibility of unauthorized origin crossing attacks on popular Android apps, such as Facebook and Dropbox. Those attacks either need to invoke the browser to load a Dialog URL (Facebook) or trick the victim user to click on a malicious link (Dropbox). However, the attacking techniques on the clipboard discussed in this section bring in another way to conduct such attacks. Malicious apps can simply replace the clipboard content with the malicious JavaScript code, which simulates a URL redirecting event to the malicious scheme. Once the code is pasted into browser's URL bar, all the attacks work the same way as in [36].

A.3.4 Dynamic Page Construction

The behavior of pure client-side web apps entirely depends on user interactions. The sanitizing technique is less likely to be applied, since the input is provided by the "trusted" device owner and will only stay within the app itself. However, if the data is copied from the infected clipboard, attackers could potentially trigger the victim apps to perform privileged operation, assuming corresponding permissions are granted to the victim app in advance.

In the study, focus has been given to PhoneGap-based apps that do not have a server side. The reason is that, as an appealing framework for developers targeting at multiple mobile platforms, PhoneGap is relatively new and few security concerns have been brought into developers' consideration. The first step of the analysis is to select candidate apps that potentially have the vulnerabilities. For that purpose, all the Android apps listed in the PhoneGap homepage have been downloaded, with exceptions of the ones requiring an



(a) Select Tags (b) Save SmartGroup

Fig. A.5.: Attack on the Vulnerable Task Manager App

account on the server side. After that, each app has been examined for web pages dynamically constructed from user input. The work can be eased with proper static JavaScript analysis tools. However, due to the dynamic feature of JavaScript as a programming language, existing static analysis tools [132, 133] are only able to serve as syntax checkers and validators. Considering the small number of the candidate apps, each one of them is analyzed manually. Finally, malicious JavaScript code is injected to target apps to determine whether they are indeed vulnerable.

One vulnerable app, called "Get It Done Task List" [134], is found in the dataset, which has roughly 50,000 installs. It is a simple but powerful to-do list and project manager, which allows each project to be assigned with a tag, and multiple tags can be managed together as a "Smart Group". When creating a smart group, the user first selects desired tags. Then the next web page is dynamically constructed with the all the selected tag names. Due to the lack of sanitizing, if the tag name comes from infected clipboard data, attackers can inject malicious JavaScript code and take advantage of all the registered JavaScript interfaces inside the victim app, as shown in Figure A.5. Considering the newly

arriving PhoneGap framework and the limited app set, the security situation of the entire Android app market may be worse in the future, if appropriate attentions are not raised on this issue.

A.3.5 SQL-Type Code Injection

In Android, web browsing within apps is eased by the WebView [135] technique, which packages basic functionalities of browsers, such as page rendering, navigation, and JavaScript execution into a class. Applications requiring these browser functionalities can simply include the WebView library and create an instance of WebView class. By doing so, apps essentially embed a basic browser in them, and can thus use it to display web contents and interact with the Web. The interaction is bidirectional: an app can register JavaScript interfaces to its WebView component so that in the future, web pages can access the app's functionalities and resources; an app can also directly load JavaScript into WebView via *loadUrl()* API. This section focuses on the risks from apps to their WebView components. However, advanced attackers can use the other interaction channel to communicate back, and thus cause damage on the app side.

The JavaScript code loaded to WebView can be pre-defined in apps' source code. Sometimes, however, the need to dynamically construct JavaScript code and load it to WebView is also legitimate. For example, an app may choose to use the following JavaScript to provide search functionality on the loaded web pages in its WebView component:

```
wv.loadUrl("javascript:search(" + input + ");");
```

In the example code, *search()* is a JavaScript API that takes user input as the search string and return its occurrence. However, the user-provided search string is not filtered for escape characters. If the user pastes the search string from the clipboard, attackers can potentially inject malicious JavaScript code into the vulnerable app, which results in manipulation of the statement running on the web pages. This attacking technique is quite similar to the well-studied SQL injection attack, in which malicious SQL statements are inserted into an entry field for execution.

JSGuard Design and Implementation. There are three key observations from the vulnerable code above. The first one is regarding the app architecture. It must have a WebView component incorporated and directly execute JavaScript code on loaded web pages. The second observation is the specific pattern of the loaded JavaScript code, which combines pre-defined code, as well as user input obtained during runtime. The last one is the lack of scrutinizing on user provided JavaScript code segment. With all the three observations in mind, an analysis tool, called JSGuard, has been developed to detect this vulnerability in Android apps on a large scale. JSGuard is based on Androguard [47], which provides rich functionalities to retrieve various app resources from its `apk` file. JSGuard totally contains 160 Lines Of Code (LOC) written in python, and its underlying logic is depicted in Figure A.6.

The input is the same app set as used in the clipboard usage analysis. In the detection phase, JSGuard first checks the existence of WebView libraries inside candidate apps. To do that, JSGuard opens each `apk` file; disassembles its `classes.dex` file and searches for WebView class from included packages. Similarly, the use of *loadUrl()* API can also be

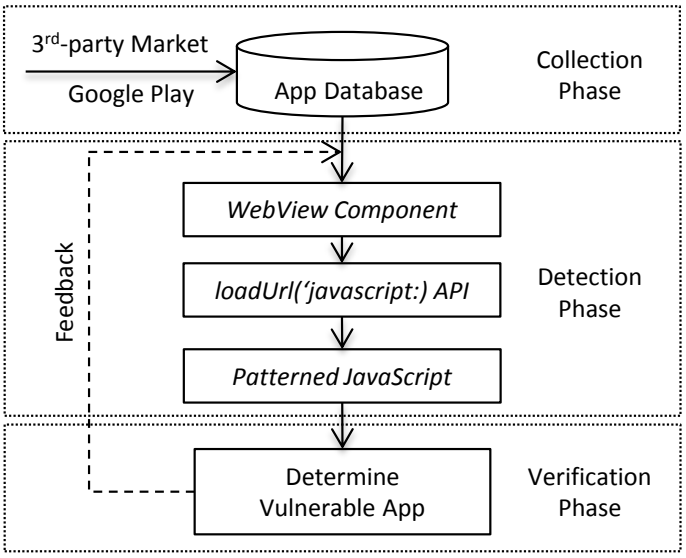


Fig. A.6.: JSGuard Design

examined. However, in order to determine whether *loadUrl()* is used to load normal web URLs or JavaScript, JSGuard has to further decompile the function in which *loadUrl()* API is invoked, extract the source code and match "javascript" with the start of *loadUrl()* argument.

Applications with JavaScript inside *loadUrl()* are not necessarily vulnerable since the JavaScript can be pre-defined. The challenge is how to detect dynamic constructed JavaScript in the static analysis. The solution comes from another observation of the decompiled source code: concatenation of String, which is achieved using "+" operator or "*concat*" API in Java, are both decompiled as "*.append()*". It should be noted that the detection algorithm so far tries to reduce the false negative as much as we can, but may mislabel secure apps. From the security perspective, however, it is more tolerable to have an absolute secure app labeled as vulnerable for future verification, rather than a vulnerable app that is considered as secure and put on the market.

Once apps are identified as containing patterned JavaScript, the potential vulnerabilities inside are manually verified by launching the SQL-type JavaScript injection attacks mentioned above. The manual verification experience can further help to improve the detection algorithm. For example, several apps are mislabeled as vulnerable because of the suspicious JavaScript code pattern inside the incorporated AdMob advertising libraries. However, the appended string comes from pre-defined advertisement settings and there is no way for attacker to inject malicious code.

Analysis Results and Case Studies. The detection phase takes around 42 hours to finish, with an average of 20 seconds spending on each app. The result shows that the use of WebView is pervasive. More than 58% of the analyzed apps also uses *loadUrl()* API to execute JavaScript code directly inside web pages. Even if only considering apps with the vulnerable JavaScript pattern, 1098 (9.4%) need further verification. In the study, 100 out of the 1098 apps are randomly selected to manually verify the existence of vulnerabilities.

Two representative vulnerable apps are found. The first one is an e-book called "Marine Martial Arts MCRP 3-02B" [136], which has roughly 500,000 installs and uses WebView to display the book content. The second one is an official Samsung app named "Smart TV Now" [137] for its Smart TV product. Currently, the app has more than 500,000 installs on Google Play market. More importantly, it is developed by Samsung developers, which are labeled as "TOP DEVELOPERS" on Google Play. Both vulnerabilities are caused of the "Search Box" inside the app, which enables user to type in the search text, and then conducts the search operation. The implementation of the search feature is identical to the example JavaScript code above. Obviously, if the victim pastes the search string from the

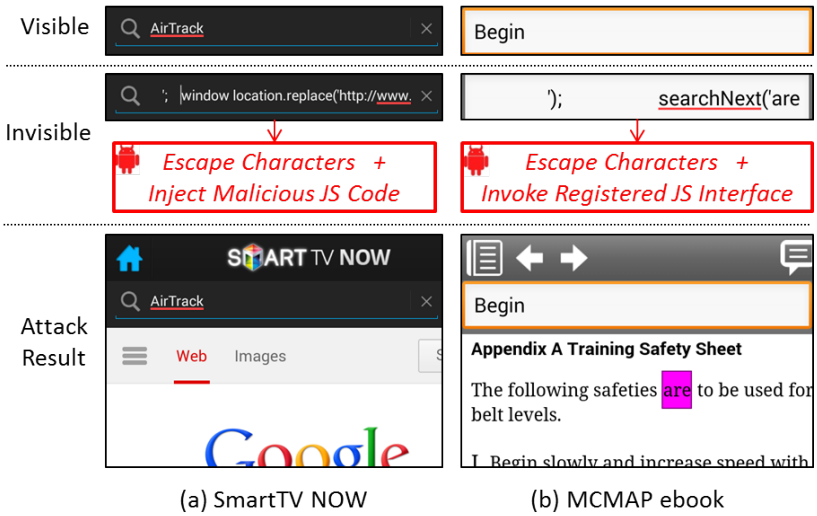


Fig. A.7.: SQL-Type Code Injection Attacks

clipboard, the attacker could potentially inject malicious JavaScript code or invoke registered JavaScript interfaces inside the app, as shown in Figure A.7.

A.4 Injection Attacks - Command

The computing power brought by mobile devices is becoming as competitive as normal desktops, but in the palm of our hands or in our pockets. Now they are not only considered as cell phones, but more of tools to help people finish complicated tasks in their daily life and in work. In Android, terminal apps are widely available on various markets. Based on provided functionalities, they usually fall into three different categories: ***Remote Terminal*** can be used to establish a connection with remote servers; ***Device Terminal*** enables the access to Android's built-in Linux command line shell; ***Combined Terminal*** incorporates both the functionalities mentioned above. Due to the general lack of physical keyboard on mobile devices and the complexity of command composition, most of terminal

Application Name	Type	# of Installs	Copy	Paste
Android Terminal Emulator	Device Terminal	5,000,000 - 10,000,000	✓	✓
ConnectBot	Remote Terminal	1,000,000 - 5,000,000	✓	✓
Android Terminal	Device Terminal	100,000 - 500,000	✗	✓
JuiceSSH - SSH Client	Combined Terminal	100,000 - 500,000	✓	✓
Terminal IDE	Combined Terminal	100,000 - 500,000	✓†	✓
Server Auditor - SSH client	Remote Terminal	10,000 - 50,000	✓	✓

† Can copy everything in the current terminal, selectively copy is not supported.

Table A.2: Study on Android Terminal Applications

apps support command copy and paste in common. However, the support is blind and the source of the pasted command is never validated. It can be either from a legitimate user copy or from the polluted copy already manipulated by attackers.

In our study, a total of six popular Android terminal apps are selected and evenly distributed to each of the three categorizes, as shown in Table A.2. Among them, *Android Terminal* [138] is the only one that does not support in-app command copy. However, there are various other sources, such as emails and websites, where victim users can copy commands. The most important observation from the study is that all the selected apps allow user to paste and execute commands within their terminals. If the pasted commands have been manipulated by malicious apps installed on the same device, depending on the type of the current connection session, various attacks can be launched against the remote server or even the Android device itself.

The damage caused by vulnerable remote terminal apps on the connected server is self-explained. Basically, attackers can potentially take full control of the remote server, steal private data or even delete all the important content. On the other hand, if malicious commands are pasted to Android Debug Bridge (`adb`) shell provided in device/combined

terminal apps, attackers can successfully perform any built-in operations, assuming the device is rooted so that each app is running with root privilege. Otherwise, attacker's capability will be restricted by the permission set granted to the victim app. Attackers can also hide themselves from user consent by appending a newline symbol and the "clear" command. While the newline symbol will force the execution of malicious commands immediately after user's paste operation, "clear" command will remove the execution history from the current terminal window.

A.5 Injection Attacks - Phishing

Phishing attacks, known as attempts to acquire sensitive information by masquerading as a trustworthy entity [139], have increased exponentially in recent years [140]. Despite common phishing techniques [141], Android Clipboard makes it easier for attackers to successfully launch phishing attacks, since mobile users perform much more copy-paste operations compared to on desktop environment, leaving attacks plenty of opportunities to redirect users to malicious entities. Based on different targets, phishing attacks on Android devices are categorized in Figure A.8.

Social Website Phishing. Entry-level attackers can simply replace all the URLs copied to the clipboard with desired ones, leading to massive advertising. The assumption is that copied URLs are always lengthy and complicated, so that it is extremely difficult for user to notice the URL differences before hitting the "Enter". However, advanced attackers may selectively replace matched URLs copied to the clipboard. In this case, even if URLs are short and easy to distinguish, attackers can leverage on some common tricks, such as

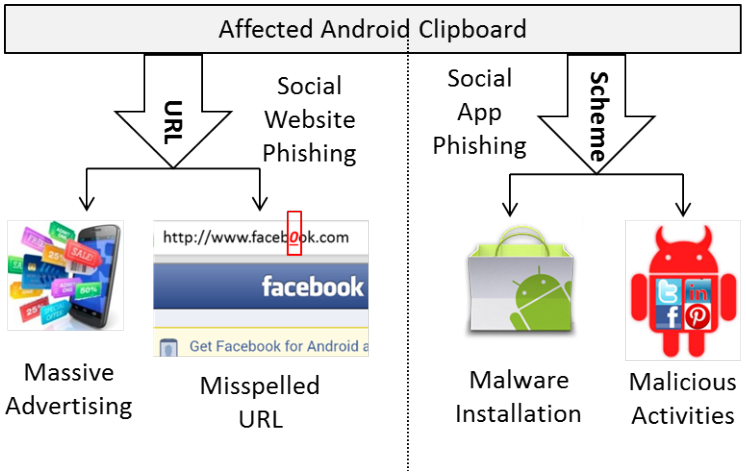


Fig. A.8.: Mobile Phishing Attacks via the Clipboard

misspelled URLs, to succeed in phishing attack. In Figure A.8, the malicious app replaces legitimate Facebook URLs with *http://www.faceb0ok.com/*. It appears as though the URL will take you to the official Facebook website; actually this URL points to the "faceb0ok" (i.e. phishing) domain which is controlled by the attacker.

Social Application Phishing. Phishing attacks on mobile platforms can also be connected with malicious apps using the scheme mechanism mentioned in Section A.3.3. Firstly, all the URLs can be replaced with Google Market scheme, tricking installation of malicious apps from victim users. Moreover, attackers can design a large number of activities in their malicious apps, with each activity representing one targeted social app's appearance. For example, any app on the device can design an activity that looks exactly like the login page in the official Facebook app. When a URL belonging Facebook domain is copied to the clipboard, that app replaces it with proper scheme that can launch its Facebook-like activity. Most likely, victim users would type in their Facebook account

Contact (3/30)	**Rank**	**# of Installs**
DW Contacts&Phone&Dialer	8	1,000,000 - 5,000,000
Contact Picker 2.3	9	5,000,000 - 10,000,000
Phone Book ConTacTs	21	100,000 - 500,000
Calendar (4/30)	**Rank**	**# of Installs**
Business Calendar Free	6	1,000,000 - 5,000,000
PETATTO CALENDAR	14	1,000,000 - 5,000,000
DigiCal Calendar&Widgets	20	500,000 - 1,000,000
Gemini Calendar	23	100,000 - 500,000
Messenger	**Rank**	**# of Installs**
ALL	N/A	N/A

Table A.3: Study on Popular Android Apps that Could Leak Sensitive Data

information, since they are expecting something happen from Facebook, either in browser or from the "Facebook" (phishing) app.

A.6 Data Leakage Attacks

Considering various types of sensitive information stored on mobile devices: once they are copied to the clipboard, malicious apps could easily steal the user's private information. In this section, case studies are conducted on three main type of sensitive data on mobile device to demonstrate the severity of the attack. For each category, the top 30 free apps on Google Play have been selected to study the possibility of sensitive data leakage. The results are summarized in Table A.3.

As the result shows, three (10%) of the studied 3rd-party Android Contact apps have the clipboard support, while four (13.3%) of the studied 3rd-party Android Calendar apps, with at least 2,600,000 installs in total, support event copying. In order to better cooperate with other apps, they all choose to resolve the Contact or event information as pure text

first and then put on the clipboard. The situation becomes even worse when it comes to messaging. All the studied messenger apps, including the built-in one on Android, allow message copying and pasting. Once the messages are copied and placed on the clipboard, malicious apps could access them without declaring the *READ_SMS* permission.

A.7 Discussion

A.7.1 Desktop Clipboard Security

Attacks caused by the clipboard on desktop environment have been observed in past few years. Self-XSS attack [116] abuses the JavaScript execution in modern browsers' address bar, and convinces a victim user to copy-paste malicious JavaScript in the URL bar and hit "Enter". Clipboard hijacking attack [142] is an exploit in which the attacker gains control of the victim's clipboard and replaces its contents with malicious data. Two solutions [117, 118] have been proposed and implemented to mitigate the problems above. However, it has been demonstrated that attackers are still able to bypass the protection on Chrome [119]. The work shown in this chapter is similar to them in exploiting vulnerabilities inside an app via the clipboard. However, it differs from them in four aspects:

Platform. The analysis presented in this chapter focuses on mobile platforms, more specifically, Android. To the best of our knowledge, this work is the first one that provides a systematic study on threats imposed by Android Clipboard. Compared to desktop

environment, mobile devices contain more sensitive data of the user. Thus, any security compromise will infer a larger damage on victim users.

Attack Efforts. The desktop clipboard is user-driven, i.e., a window will transfer data to or from the clipboard only in response to a command from the user. To carry out the attacks on Desktop, significant social engineering efforts are involved to trick victim users to conduct desired operations. In contrast, any apps installed on the same Android mobile device potentially can launch the attack without requiring any special privileges.

Attack Surface. The attacking surface on mobile devices is larger than on desktop. The attacks on the desktop clipboard only target at browser or web-based apps. However, in this work, many other apps, such as terminal apps, Contacts apps, Calendar apps and etc., have been demonstrated to be vulnerable to attacks through Android Clipboard.

Solutions. Google, Firefox, Adobe and other big companies have taken the lead to fix the clipboard problem on desktop environment. However, equivalent efforts are missing on mobile platforms, considering that only two out of the top eleven Android browsers restrict JavaScript execution in URL bar. Moreover, existing solutions on desktop environment are limited to specific apps [117, 118].

A.7.2 Improving Android Clipboard Security

Unlike the desktop environment, Android treats each app as a different user with different privileges. However, a similar design for the desktop clipboard is blindly moved to the Android platform without corresponding changes to accommodate its different security

model. In this section, several potential solutions will be discussed from different perspectives for protecting Android Clipboard from being abused. While implementation details are omitted, each proposed mechanism could be easily integrated into the existing Android framework.

From the User Perspective. In the current Android implementation, when the user copies data into the clipboard, an alert is displayed. However, such an alert message is missing when an app silently manipulates or steals the data using the clipboard APIs. A similar warning message, which displays the calling app's information, may help users detect malicious apps' suspicious behaviors. Then the user can either refuse to paste the injected data from the clipboard, or simply uninstall the calling app. This protection, however, is passive, which solely depends on users' awareness of security and privacy.

From the Developer Perspective. There is always a battle between app features and the security consideration. For example, three studied Contacts apps add the integral Contact copy feature to enrich their functionalities, and thus, attract more users. However, the security is compromised since they accidentally leak private data to malicious apps. It is challenging to ask app developers to sacrifice even one feature for security enhancement. In the specific clipboard case, to protect themselves from the injection attacks, developers should always conduct validation on fields that can take input from the clipboard.

From the Clipboard Service Perspective. Android provides several system services, through which apps can access private resources or privileged system APIs. In order to access one system service, corresponding permissions have to be granted by user upon app's

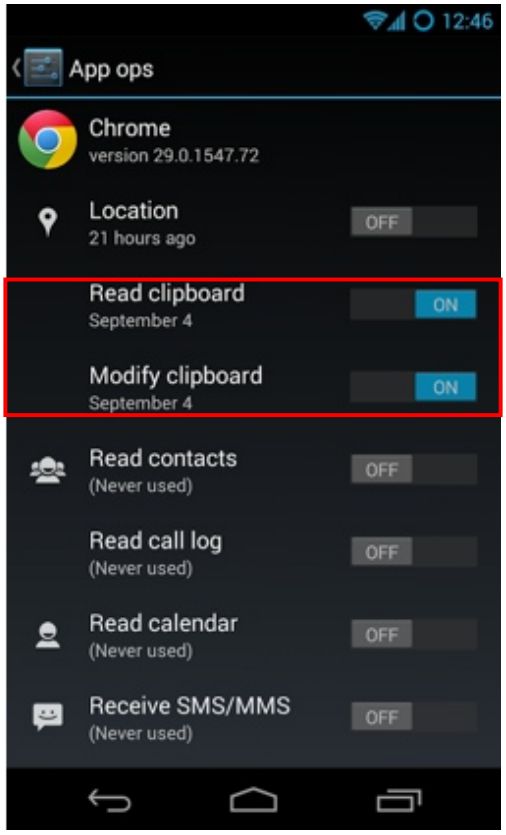


Fig. A.9.: Restricting Access to Android Clipboard via App Ops

installation. The clipboard service on Android, as another system service, however, does not require any permissions. To prevent attackers from freely accessing or manipulating the Clipboard data, three new permissions can be added: *READ_CLIPBOARD*, *WRITE_CLIPBOARD* and *MONITOR_CLIPBOARD*. Each of them corresponds to an operation supported by the clipboard service. Based on Android's permission system, user can make a decision on whether to install apps that can operate on the clipboard.

Actually, Android attempted to provide such a security improvement, namely App Ops, in version 4.3. App Ops is the hidden app permission manager that allows users to selectively disable some permissions for their apps. There are 4 permission groups supported by App Ops: location, personal (contacts, calendar, call logs), messaging

(read/write/send SMS) and device (notifications, camera). The access to Android Clipboard belongs to the personal permission group. App Ops defines two additional operations for Android Clipboard, including *OP_READ_CLIPBOARD* and *OP_WRITE_CLIPBOARD*. As shown in Figure A.9, a user can disable certain apps' access to the clipboard data at runtime. However, Google has removed all possibilities to start the App Ops on non-rooted Android 4.4.2 devices, as per-permission toggle could prevent apps from functioning properly. Admittedly, certain operations, such as sending SMS and accessing contacts, are critical to apps' functionalities, however, the access to Android Clipboard does not belong to this category. We argue that, a simplified version of App Ops specifically designed for Android Clipboard can mitigate the risks identified in this chapter without affecting apps' behaviors.

From the Android System Perspective. SEAndroid [16] and FlaskDroid [17], both proposed a flexible Mandatory Access Control (MAC) framework for Android. One advantage of MAC is the ability to confine privileged Android system daemons and access to system resources by apps. By extending their policy enforcement, access to the clipboard service can be restricted to certain apps.

A.8 Conclusion

In this chapter, we have assessed the current security situation of Android Clipboard by examining its usage in 16,000 benign apps and 3,987 malicious apps. Based on the risk assessment, we have formulated a series of attacks and categorize them into two groups, i.e., manipulation and stealing. Clipboard data manipulation may lead to code injection

attacks and phishing attacks. Data stealing happens when sensitive data or reference is copied to the clipboard. The presence of vulnerable apps as well as a variety of attack types reflects the severity of the risks imposed by Android Clipboard. As a result of that, we suggest developers to be cautions of dealing with the clipboard data.

LIST OF REFERENCES

[1] "IDC Report." `http://goo.gl/aPfSKz`.

[2] "AppBrain Statistic on Android Apps." `http://goo.gl/CyDYNc`.

[3] "Statista Report." `http://goo.gl/kkwLW9`.

[4] X. Zhang, K. Ying, Y. Aafer, Z. Qiu, and W. Du, "Life after app uninstallation: Are the data still alive? data residue attacks on android," in *NDSS*, 2016.

[5] 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), pp. 101–112, ACM, 2012.

[6] 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), pp. 2:1–2:6, ACM, 2012.

[7] 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), pp. 346–360, Springer-Verlag, 2011.

[8] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: attacks and defenses," in *Proceedings of the 20th USENIX conference on Security symposium*, 2011.

[9] 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 2012, San Diego, California, USA, February 5-8, 2012*, 2012.

[10] Y. Zhou and X. Jiang, "Detecting passive content leaks and pollution in android applications," in *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*, 2013.

[11] M. Zhang and H. Yin, "Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications," in *NDSS*, 2014.

[12] 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), pp. 239–252, ACM, 2011.

[13] 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.

[14] 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), pp. 125–136, ACM, 2012.

[15] M. Backes, S. Bugiel, and S. Gerling, "Scippa: System-centric ipc provenance on android," in *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, (New York, NY, USA), pp. 36–45, ACM, 2014.

[16] S. Smalley and R. Craig, "Security enhanced (SE) android: Bringing flexible MAC to android," in *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*, 2013.

[17] 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.), pp. 131–146, USENIX, 2013.

[18] 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.

[19] "Google AdMob Ads SDK." `https://goo.gl/FM43NG`.

[20] M. Nauman, S. Khan, and X. Zhang, "Apex: extending android permission model and enforcement with user-defined runtime constraints," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, 2010.

[21] Y. Wang, S. Hariharan, C. Zhao, J. Liu, and W. Du, "Compac: Enforce component-level access control in android," in *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, CODASPY '14, (New York, NY, USA), pp. 25–36, ACM, 2014.

[22] 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*, 2012.

[23] 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), pp. 28–28, USENIX Association, 2012.

[24] T. Luo, X. Jin, A. Ananthanarayanan, and W. Du, "Touchjacking Attacks on Web in Android, iOS, and Windows Phone," in *Proceedings of the 5th International Symposium on Foundations & Practice of Security*, October 25–26 2012.

[25] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in *Proceedings of the 20th USENIX conference on Security*, SEC'11, (Berkeley, CA, USA), pp. 21–21, USENIX Association, 2011.

[26] M. Wei, D. Ren, P. C. Simon, H. Steven, and L. Wenke, "The price of free: Privacy leakage in personalized mobile in-app ads," in *NDSS*, 2016.

[27] W. Enck, M. Ongtang, and P. McDaniel, "Understanding android security," *Security Privacy, IEEE*, vol. 7, pp. 50–57, Jan 2009.

[28] 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), pp. 627–638, ACM, 2011.

[29] P. Ratazzi, Y. Aafer, A. Ahlawat, H. Hao, Y. Wang, and W. Du, "A systematic security evaluation of Android's multi-user framework," in *Mobile Security Technologies (MoST) 2014*, MoST'14, (San Jose, CA, USA), May 17 2014.

[30] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang, "The peril of fragmentation: Security hazards in android device driver customizations," in *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA*.

[31] R. Gallo, P. Hongo, R. Dahab, L. C. Navarro, H. Kawakami, K. Galvão, G. Junqueira, and L. Ribeiro, "Security and system architecture: Comparison of android customizations," in *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, WiSec '15, (New York, NY, USA), pp. 12:1–12:6, ACM, 2015.

[32] 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), pp. 623–634, ACM, 2013.

[33] 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), pp. 81–92, ACM, 2012.

[34] M. Zhang, Y. Duan, Q. Feng, and H. Yin, "Towards automatic generation of security-centric descriptions for android apps," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, (New York, NY, USA), pp. 518–529, ACM, 2015.

[35] 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), pp. 343–352, ACM, 2011.

[36] 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), pp. 635–646, ACM, 2013.

[37] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri, "Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, (New York, NY, USA), pp. 66–77, ACM, 2014.

[38] 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), pp. 73–84, ACM, 2013.

[39] 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), pp. 659–668, ACM, 2013.

[40] 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), pp. 393–408, IEEE Computer Society, 2014.

[41] E. Fragkaki, L. Bauer, L. Jia, and D. Swasey, "Modeling and enhancing androidâĂŹs permission system," in *Computer Security âĂŞ ESORICS' 12*, vol. 7459 of *Lecture Notes in Computer Science*, 2012.

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

[43] 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.

[44] 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.

[45] 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.

[46] X. Zhang and W. Du, "Attacks on android clipboard," in *Detection of Intrusions and Malware, and Vulnerability Assessment* (S. Dietrich, ed.), vol. 8550 of *Lecture Notes in Computer Science*, pp. 72–91, Springer International Publishing, 2014.

[47] "AndroGuard." `http://code.google.com/p/androguard/`.

[48] 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), pp. 229–240, ACM, 2012.

[49] 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), pp. 259–269, ACM, 2014.

[50] 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.), pp. 543–558, USENIX, 2013.

[51] "WALA." `http://wala.sourceforge.net/`.

[52] "Soot." `http://sable.github.io/soot/`.

[53] W. Enck, P. Gilbert, S. Han, V. Tendulkar, 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," *ACM Trans. Comput. Syst.*, vol. 32, pp. 5:1–5:29, June 2014.

[54] 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), pp. 639–652, ACM, 2011.

[55] "SCanDroid." `http://spruce.cs.ucr.edu/SCanDroid/`.

[56] 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), pp. 235–245, ACM, 2009.

[57] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, "Semantically rich application-centric security in android," in *Proceedings of the 2009 Annual Computer Security Applications Conference*, 2009.

[58] 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), pp. 27–27, USENIX Association, 2012.

[59] H. Hao, V. Singh, and W. Du, "On the effectiveness of api-level access control using bytecode rewriting in android," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, (New York, NY, USA), pp. 25–36, ACM, 2013.

[60] A. Viswanathan and B. Neuman, "A survey of isolation techniques,"

[61] F. B. Schneider, J. G. Morrisett, and R. Harper, "A language-based approach to security," in *Informatics - 10 Years Back. 10 Years Ahead.*, (London, UK, UK), pp. 86–101, Springer-Verlag, 2001.

[62] "Android ndk." `http://developer.android.com/tools/sdk/ndk/index.html#Samples`.

[63] M. Sun and G. Tan, "Nativeguard: Protecting android applications from third-party native libraries," in *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks*, WiSec '14, (New York, NY, USA), pp. 165–176, ACM, 2014.

[64] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets," in *19th Annual Network and Distributed System Security Symposium*, 2012.

[65] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky, "Appguard: Enforcing user requirements on android apps," in *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'13, (Berlin, Heidelberg), pp. 543–548, Springer-Verlag, 2013.

[66] A. Bartel, J. Klein, M. Monperrus, K. Allix, and Y. L. Traon, "Improving privacy on android smartphones through in-vivo bytecode instrumentation," *CoRR*, vol. abs/1208.4536, 2012.

[67] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen, "I-arm-droid: A rewriting framework for in-app reference monitors for android applications," *Mobile Security Technologies*, vol. 2012, 2012.

[68] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, (New York, NY, USA), pp. 203–216, ACM, 1993.

[69] 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 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

[70] D. Ehringer, "The dalvik virtual machine architecture," 2010.

[71] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh, "Cells: A virtual mobile smartphone architecture," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, (New York, NY, USA), pp. 173–187, ACM, 2011.

[72] C. Wu, Y. Zhou, K. Patel, Z. Liang, and X. Jiang, "Airbag: Boosting smartphone resistance to malware infection.," 2014.

[73] "Same-origin policy." `http://goo.gl/P4OVoN`.

[74] "HTML <iframe> sandbox Attribute." `http://www.w3schools.com/tags/att_iframe_sandbox.asp`.

[75] "The Chromium Projects." `https://www.chromium.org/`.

[76] "Out-of-Process iframes (OOPIFs)." `https://www.chromium.org/developers/design-documents/oop-iframes`.

[77] "Site Isolation." `https://www.chromium.org/developers/design-documents/site-isolation`.

[78] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, 2009.

[79] "Localytics on App Retention." `http://goo.gl/NWRCGL`.

[80] "iResearch on App Life Expectancy." `http://goo.gl/jwENYX`.

[81] "AndroidCentral Poll." `http://goo.gl/nl5z6y`.

[82] "AccountManager Changelog." `https://goo.gl/oD2qXt`.

[83] "ADAL Android SDK." `https://goo.gl/f79B4k`.

[84] "App Genome Report." `https://goo.gl/eGszpB`.

[85] "Complement in set theory." `https://goo.gl/vlpHym`.

[86] "dextra - A tool for DEX and OAT dumping, decompilation, and fuzzing." `http://goo.gl/NPG0Kz`.

[87] "Lollipop deodexing." https://goo.gl/uw2KmR.

[88] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, "Edgeminer: Automatically detecting implicit control flow transitions through the android framework," in *NDSS*, 2015.

[89] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis," in *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, (Berkeley, CA, USA), pp. 543–558, USENIX Association, 2013.

[90] "Java: Computing Cyclomatic Complexity." http://goo.gl/tduqlP.

[91] "Cyclomatic complexity." https://goo.gl/1VqYUj.

[92] "Java Varargs." http://goo.gl/TEMrjk.

[93] D. F. Bacon, "Fast and effective optimization of statically typed object-oriented," tech. rep., Berkeley, CA, USA, 1998.

[94] D. F. Bacon and P. F. Sweeney, "Fast static analysis of c++ virtual function calls," in *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, (New York, NY, USA), pp. 324–341, ACM, 1996.

[95] D. Grove, G. DeFouw, J. Dean, and C. Chambers, "Call graph construction in object-oriented languages," in *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, (New York, NY, USA), pp. 108–124, ACM, 1997.

[96] O. G. Shivers, *Control-flow Analysis of Higher-order Languages of Taming Lambda*. PhD thesis, Pittsburgh, PA, USA, 1991. UMI Order No. GAX91-26964.

[97] "ext4_utils." https://goo.gl/1nyYfM.

[98] "A tool for converting android's .dex format to java's .class format." http://code.google.com/p/dex2jar.

[99] "Apktool: A tool for reverse engineering Android apk files." http://goo.gl/LdB4V7.

[100] "smali and baksmali." https://goo.gl/JS7Mgw.

[101] "ANRED: Android Residue Detection Framework." https://goo.gl/Q0d5qH.

[102] "Huawei ROMs." http://goo.gl/dYPTE5.

[103] "Factory Images for Nexus Devices." https://goo.gl/i0RJnN.

[104] "Official Oxygen OS ROMs and OTA updates." https://goo.gl/cBTF1w.

[105] "Cyanogenmod Downloads." http://download.cyanogenmod.org/.

[106] "Android Revolution." http://goo.gl/MVigfq.

[107] "Amazon Fire OS." https://en.wikipedia.org/wiki/Fire_OS.

[108] A. Farahat and M. C. Bailey, "How effective is targeted advertising?," in *Proceedings of the 21st international conference on World Wide Web*, WWW '10, 2012.

[109] D. BELIC, "Three things to know about mobile advertising in 2011." `http://goo.gl/XAfZXU`.

[110] "Skymap." `https://goo.gl/B91B6c`.

[111] "Appbrain." `http://www.appbrain.com/`.

[112] H. Davis, "Open source android applications." `http://goo.gl/EzwcAq`.

[113] "Antutu benchmark." `https://goo.gl/IkFlTR`.

[114] "Android memory usage." `http://elinux.org/Android_Memory_Usage`.

[115] "About the Clipboard." `http://goo.gl/db36Wd`.

[116] "Self-XSS Attack Explained." `https://goo.gl/ssgTI9`.

[117] "Firefox Disallows javascript in its URL Bar." `https://goo.gl/UJ8AIg`.

[118] "Pasting a javascript: url from the omnibar removes the protocol." `http://goo.gl/tuY9Gq`.

[119] "Self XSS protection bypass to paste and execute Javascript in the address-bar." `https://goo.gl/GdWLbx`.

[120] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: Mining API Level Features for Robust Malware Detection in Android," in *Proceedings of the 9th International Conference on Security and Privacy in Communication Networks (SecureComm)*, (Sydney, Australia), September 25-27 2013.

[121] "Android Malware Genome Project." `http://www.malgenomeproject.org/`.

[122] "phpBB." `https://www.phpbb.com/`.

[123] "HttpOnly." `https://www.owasp.org/index.php/HttpOnly`.

[124] "Phonegap: Easily create apps using the web technologies you know and love: Html, css and javascript." `http://phonegap.com`.

[125] P. Bisht and V. N. Venkatakrishnan, "XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks," in *DIMVA 2008*.

[126] M. Johns, "SessionSafe: Implementing XSS Immune Session Handling," in *ESORICS 2006*.

[127] M. Martin and M. S. Lam, "Automatic Generation of XSS and SQL Injection Attacks with Goal-Directed Model Checking," in *USENIX-SS 2008*.

[128] M. Ter˜Louw, P. Bisht, and V. N. Venkatakrishnan, "Analysis of Hypertext Isolation Techniques for {XSS} Prevention," in *Web 2.0 Security and Privacy 2008*, May 2008.

[129] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *ICSE 2008*.

[130] "Android Scheme." `http://goo.gl/ZAAqj1`.

[131] "iOS SDK: Working with URL Schemes." http://goo.gl/gMl3cW.

[132] "JSLint." http://www.jslint.com/.

[133] "JSure." https://github.com/berke/jsure.

[134] "Get It Done Task List." https://goo.gl/lK0zmN.

[135] Android-Team, "Webview class reference." http://goo.gl/uzqdt1.

[136] "Marine Martial Arts MCRP 3-02B." https://goo.gl/Lb8rXF.

[137] "Samsung Smart TV Now." https://play.google.com/store/apps/details?id=com.samsung.videocloud.

[138] "Android Terminal." https://goo.gl/dox0ov.

[139] "Phishing." http://en.wikipedia.org/wiki/Phishing.

[140] "RSA's October Online Fraud Report 2012 including summary of Phishing and Social Networking." http://goo.gl/m7JN5e.

[141] "Phishing Techniques." http://www.phishing.org/phishing-techniques/.

[142] "Clipboard Hijack Attack." http://goo.gl/L3BwzW.

VITA

Xiao Zhang

Syracuse University Graduate Department of Computer Science

xzhang35@syr.edu

114 Janet Drive, Syracuse, NY, 13224

Education

- **Ph.D. in Computer Science**
  Syracuse University, USA, 2016
- **Bachelor of Science in Computer Science**
  University of Science and Technology of China, China, 2010

.

EXPERIENCE

- **Syracuse University.**, 01/15 - 05/16, **Research Assistant:** Conduct Research on Android data residue vulnerability and hanging attribute (Hare) references vulnerability.
- **SEED Workshop.**, 01/2015 - 06/2015, **Organizer:** Help organize SEED workshop in ACM Technical Symposium on Computer Science Education (SIGCSE 2015) and in Syracuse University during June 2015.
- **Samsung Research America, Knox Team.**, 09/14 - 01/15, **Research Intern:** Design and implement Single Sign On feature on Samsung Knox platform for enterprise mobile security. Involve in the preparation of pattern titled "SYSTEM AND METHOD FOR A GENERIC SINGLE SIGN-ON FUNCTION".
- **Cigital, Inc.**, 05/14 - 08/14, **Security Consultant Intern:** Conduct both static analysis and dynamic analysis testing on various client products, including Android/iOS/web applications and source code review.

- **Syracuse University.**, 09/10 - 05/14, **Teaching Assistant:** Help instructors organize classes and lead lab sessions, including both graduate level courses (Internet Security, Computer Security, Principles of Operating System, Algorithm) and undergraduate level courses (Programming Language, Data Structures).

AWARDS

- **Practical Application Winner**: NUNAN Research Competition, EECS Dept., Syracuse University, April 15, 2015.
- **Department Winner**: NUNAN Research Competition, EECS Dept., Syracuse University, April 13, 2012.

PUBLICATIONS

---

1. **Hey, You, Get Off of My Image: Detecting Data Residue in Android Images**,
   X. Zhang, Y. Aafer, K. Ying and W. Du,
   (under submission)
2. **Harvesting Inconsistent Security Configurations in Custom Android ROMs via Differential Analysis**,
   Y. Aafer, X. Zhang, and W. Du,
   (under submission)
3. **Life after App Uninstallation: Are the Data Still Alive? Data Residue Attacks on Android**,
   X. Zhang, K. Ying, Y. Aafer, Z. Qiu and W. Du,
   in Network and Distributed System Security Symposium, NDSS '16, (San Diego, CA, USA), Feb. 21-24, 2016.
4. **Hare Hunting in the Wild Android: A Study on the Threat of Hanging Attribute References**,
   [Y. Aafer, N. Zhang]co-first author, Z. Zhang, X. Zhang, K. Chen, X. Wang, X. Zhou, W. Du, and M. Grace,
   in Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, CCS 15', (Denver, Colorado, USA), Octobor. 12-16, 2015.
5. **Attacks on Android Clipboard**,
   X. Zhang and W. Du,
   in Proceedings of the 11th Conference on Detection of Intrusions and Malware & Vulnerability Assessment, DIMVA '14, (Egham, UK), July 10-11, 2014.
6. **AFrame: Isolating Advertisements from Mobile Applications in Android**,
   X. Zhang, A. Ahlawat and W. Du,
   in Proceedings of the 29th Annual Computer Security Applications Conference, ACSAC '13, (New Orleans, Louisiana, USA), December 9-13, 2013.