

ABSTRACT

The pervasiveness of Android devices in today’s interconnected world emphasizes the importance of mobile security in protecting user privacy and digital assets. Android’s current security model primarily enforces application-level mechanisms, which fail to address component-level (e.g., Activity, Service, and Content Provider) security concerns. Consequently, third-party code may exploit an application’s permissions, and security features like MDM or BYOD face limitations in their implementation.

To address these concerns, we propose a novel Android component context-aware access control mechanism that enforces layered security at multiple Exception Levels (ELs), including EL0, EL1, and EL3. This approach effectively restricts component privileges and controls resource access as needed. Our solution comprises Flasa at EL0, extending SELinux policies for inter-component interactions and SQLite content control; Compac, spanning EL0 and EL1, which enforces component-level permission controls through Android runtime and kernel modifications; and TzNfc, leveraging TrustZone technologies to secure third-party services and limit system privileges via Trusted Execution Environment (TEE).

Our evaluations demonstrate the effectiveness of our proposed solution in containing component privileges, controlling inter-component interactions and protecting component level resource access. This enhanced solution, complementing Android’s existing security architecture, provides a more comprehensive approach to Android security, benefiting users, developers, and the broader mobile ecosystem.

FINE-GRAINED ACCESS CONTROL ON ANDROID COMPONENT

by

Yifei Wang

B.S., Sun Yat-sen University, 2008

M.S., Syracuse University, 2011

Dissertation

Submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer & Information Science and Engineering

Syracuse University

August 2023

Copyright © Yifei Wang 2023

All Rights Reserved

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to all those who have supported me during the course of my thesis.

My deepest gratitude goes to my advisor, Prof. Wenliang Du, for his guidance, mentorship, and constant encouragement. His invaluable insights and expertise in the security research field enabled me to delve so deeply into my research and ultimately complete this dissertation. The knowledge, attitude, and skills I learned from him has been left a long lasting impact on both my research and career.

I am also grateful to Prof. C.Y. Roger Chen, Prof. Endadul Hoque, Prof. Fanxin Kong, Prof. Quinn Qiao, and Prof. Yuzhe Tang for their willingness to serve on my thesis committee. Furthermore, I extend my thanks to the department chair, Prof. Jae C. Oh, for his care and support.

Friendships and collaborations have been invaluable throughout this long doctoral journey. I have been fortunate to collaborate with colleagues at Samsung, including Dr. Xinwen Zhang, Dr. Peng Ning, Dr. An Liu, Dr. Qian Feng, and Dr. Pai Peng. I also appreciate the opportunity to conduct research alongside my Syracuse University colleagues Chenxi Zhao, Srinivas Hariharan, Haichao Zhang, and Jiaming Liu. Additionally, I am also thankful the enrich discussions and study shared with colleagues, especially Dr. Tongbo Luo, Dr. Xiao Zhang, Dr. Kailiang Ying, Dr. Mu Zhang, Dr. Amit Ahlawat, Dr. Yousra Aafer and Dr. Paul Ratazzi.

Finally, and most importantly, I am immensely grateful to my family, especially my mother, and my wife Dr. Xiao, for their unwavering support and love.

To everyone who has helped me overcome this long research journey, I am eternally appreciative.

TABLE OF CONTENTS

	Page
ABSTRACT	i
LIST OF TABLES	x
LIST OF FIGURES	xi
1 Introduction	1
1.0.1 System-level security: application sandboxing	1
1.0.2 Application-level security: Android permission model	2
1.0.3 Limitations of application level access control	3
1.0.4 Android Component’s communication channel	4
1.0.5 Thesis Statement	5
2 Related Work	8
2.0.1 Sandboxing	9
2.0.2 Virtualization	9
2.0.3 ByteCode rewriting	10
2.0.4 TrustZone technology	10
2.0.5 Mandatory Access Control	11
3 A General Security Architecture for Diverse Mandatory Access Control Policies on Android	13
3.1 Introduction	13
3.2 Background	17
3.3 Problem Statement	19
3.3.1 Defects of the Android Permission Model	19
3.3.2 MAC for Android	21
3.4 Overview of Flasa	24
3.5 Access Control Model	26

	Page
3.5.1 Subjects and Objects	26
3.5.2 Access Modes	27
3.5.3 Labels	28
3.5.4 Policy	29
3.6 Architecture	31
3.6.1 Architecture Overview	31
3.6.2 Component Labeling	34
3.6.3 Data Labeling	34
3.6.4 Policy Enforcement	37
3.7 Implementation	37
3.7.1 Android Component Labelling	40
3.8 Limitations	42
3.9 Evaluation	44
3.9.1 Enforcing BYOD Policies	44
3.9.2 Preventing Malware	47
3.9.3 Performance	48
3.10 Related Work	50
3.11 Conclusions	53
4 Compac: Enforce Component-Level Access Control in Android	54
4.1 Introduction	54
4.2 Background: Access Control in Android	56
4.2.1 The “Windows” on the Sandbox	56
4.2.2 Reference Monitors	57
4.3 The Compac Design	58
4.3.1 The Overview	58
4.3.2 Assumption & Trusted Computing Base	60
4.3.3 Component Permission Configuration	62
4.3.4 Extracting Component Call Chains	64
4.3.5 Access Control Enforcement Based on the Call Chain	68

	Page
4.3.6 Policy Manager Implementation	70
4.4 Potential Attacks & Defense	71
4.4.1 Implicit Invocation Attack	72
4.4.2 Inter-Component Code Injection Attacks	73
4.4.3 Package Forgery “Attack”	75
4.5 Case Studies & Evaluation	75
4.5.1 Advertising APIs	75
4.5.2 Social Networking Service APIs	77
4.5.3 Web Applications - PhoneGap	79
4.5.4 Performance	81
4.6 Related Work	84
4.7 Conclusion	87
5 Understanding and Mitigating Security Risks in Host Card Emulation (HCE) . .	88
5.1 Introduction	88
5.2 Background	93
5.3 OVERVIEW	97
5.4 The HCSH attack	99
5.4.1 Attacking Analysis	99
5.4.2 Binding to a HCE Service	102
5.4.3 Bypassing individual wallet’s Authentication	104
5.4.4 Crafting Command APDUs	105
5.4.5 Replicating a Transaction Session	107
5.5 Mitigation	109
5.5.1 Adversary Model	109
5.5.2 Design Analysis	110
5.5.3 Design	111
5.5.4 Implementation	114
5.5.5 Evaluation	117
5.6 Related Work	119

	Page
5.6.1 Attacks and Solutions on NFC and EMV protocols	119
5.6.2 Attacks and Protections in Android Payment System	120
6 Summary	122
LIST OF REFERENCES	123
VITA	132

LIST OF TABLES

Table	Page
3.1 Access modes in Flasa	27
3.2 Code modification statistics	38
3.3 RL benchmark results	50
4.1 Implicit Invocation Usage (Total App Samples: Benign 16000, Malware 2566)	72
4.2 Reflection for Code Injection Usage (Total App Samples: Benign 16000, Malware 2566)	74

LIST OF FIGURES

Figure	Page
1.1 Resource Access Path of a Component	4
1.2 Solution Stack Overview	6
2.1 Related Work	8
3.1 Defects of the Android permission model.	20
3.2 Flasa overview. Flasa enforces MAC policies orthogonal to the Android permission model.	24
3.3 Flasa architecture. AMS and LMS consists of the reference monitor of Flasa. .	33
3.4 Data labeling and query. Activity with label l_1 can query data entries with label l_1 and l_3 in the database of the content provider.	36
3.5 Content provider implementation details	40
3.6 Activity invocation sequence. AMS enforces security policies when it identifies the subject and object activities.	41
3.7 BYOD case study. Flasa enables a personal application to invoke the activity of an enterprise application. By transiting the label of the privileged component, this does not leak any sensitive data.	44
3.8 Preventing malware. Flasa silently denies the malicious actions of <code>jSMShider</code> . .	47
3.9 Preventing malware. Flasa silently denies deleting SMS message from <code>HippoSMS</code> . .	48
3.10 AnTuTu benchmark results.	49
4.1 The Architecture of Compac	59
4.2 Permission Manager Interface for Users: third-party components with less permissions will be displayed in system settings, and experienced users have options to adjust component permissions in the app info section of system settings. . .	64
4.3 Policy Manager (PM) Structure & Permission Checking Flow	65
4.4 Component Intersection Cases	70
4.5 Angry Birds	76
4.6 Angry Birds: No Permission Pop-up Window	77

Figure	Page
4.7 Ads in Angry Birds: Permission Deny Logcat	77
4.8 Facebook: Permission Deny Toast	78
4.9 Facebook: Permission Deny Logcat	79
4.10 Overall Performance: Benchmark Results	79
5.1 HCE Architecture Overview	96
5.2 The HCSH attack flow	101
5.3 Bind Service	103
5.4 NCI Core Packet Format	117

1. INTRODUCTION

Android has been dominating global smartphone market since 2017, taking roughly 69.74% market share as of January 2022. As application fundamentals, Android components are the essential building blocks of an Android app. As Android developers' guides mentioned, each component is an entry point through which the system or a user can enter your app. Consequently, the access control of components is crucial to protecting users' privacy and preventing unwanted accesses.

1.0.1 System-level security: application sandboxing

Android has utilities Linux kernel sandbox to provide resource isolation. The kernel builds sandboxes based on User Id (UID), and a process can only access resources under the same UID. Each app is assigned a different UID when installed, and its process, when created, runs with the same UID, which achieves isolation among applications. Moreover, Android segregates system resources under a set of system reserved UIDs such as **system**, **nfc**, and **bluetooth**.

Such a sandbox build strong an isolation on top of Linux Kernel. Applications, however, should be allowed to access system and other apps' resources. Meanwhile, such accesses should be controlled, not arbitrary. To allow controlled accesses, "windows" have to be opened on the sandbox, but behind each window, there should be an access control.

Android opens up several types of windows, and these windows will be discussed in details later. Before that, the priority is to talk about the controls of the access through these windows. Android build such control with the help of *Android permissions*.

1.0.2 Application-level security: Android permission model

To allow controlled access to system resources and other apps' resources, Android uses the Android permission model. An application must declare Android permissions corresponding to the privileged resources it needs to access beyond its sandbox in the `AndroidManifest.xml` file. Permissions are divided into three types: install-time permissions, runtime permissions and special permissions. for install-time permissions, a system dialog is popped up to ask for the authorization. Resources guarded by such permissions are usually considered as insensitive data. Resources protected by runtime permissions are treated as potentially more privacy and security related resources, and it will pop up a window during runtime when the related resources like camera, microphone, geolocation. As of now, the stock AOSP system has defined more than 300 permissions in total of the two types. In addition, special permissions can be defined by platform or OEMs to protect access to particularly powerful actions. Some permission's' implementation like accessing camera, microphone or internet permissions, behind the scene, relies on the Linux sandboxing mechanism. For example, the internet UID will be added to the Group IDs (GIDs) of a requesting app, so that this app's process is capable of accessing the internet.

1.0.3 Limitations of application level access control

Privileged resources are mostly protected by Android permissions model, however, such controlled access is executed at app level, instead of the principal subject - Android component, which can be problematic. Firstly, an app may contain the code with different trust level because they are from multiple origins. Third Party Libraries (TPLs) like Ads and 3D graphic are widely embedded in mobile apps. Some TPLs may not as trustworthy as the app including TPLs downloaded from less trusted sites. Those TPLs may be repackaged, and untrusted code may sneak in. Even with so many TPLs at wild, Android blindly trust those TPLs and treat them the same trust level as the app. Secondly, advanced features like Bring Your Own Device (BYOD) and other advanced Mobile Device Management (MDM) require a fine grained access control to limit some of an app feature while using in certain workspace environment. Even though some components may work individually, the permission model cannot support the context of component. Therefore, MDM has to only work at app level which limits the flexibility and functionality of those features.

Thirdly, not only app but also system's privilege separation is required for least privilege principal. Android, as a complex OS, sharing a huge system context. All system components sharing similar privileges can access sensors, files or even 3rd party services.

These real world concerns and challenges lead to a research problem that how to make a further granular level of access control inside an app. The granularity of principal subject can be a process or thread at runtime, Android component, Java class, snippet of code, or

a newly proposed concept of component, etc, and the protected object should include all the possible channels of a principal subject.

1.0.4 Android Component's communication channel

App components are the essential building blocks of an app. An app may have four types of component: Activities, Services, Broadcast receivers, Content providers. An Activity is the entry point of an app to interact with a user. It represents a screen of user interface. A service is designed to run at the background for various purposes. A broadcast receiver can register system or app defined broadcast and listen to the event. A content provider provides a set of unified APIs to access database, file or other content. In general, a component, as the principal subject to interact with others and access resources can have two categories of accesses:

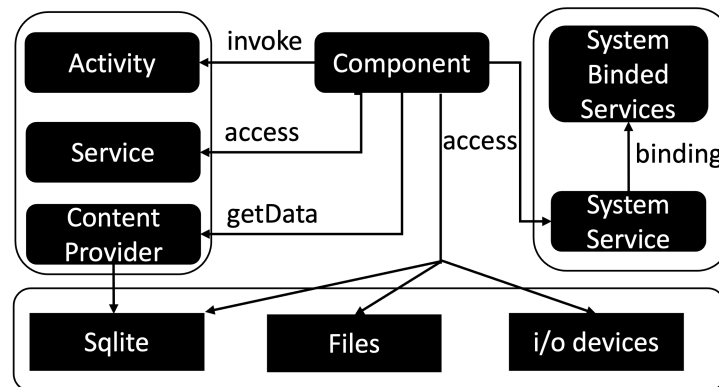


Fig. 1.1.: Resource Access Path of a Component

- A component can activate another component as mentioned. Since components are designed as the principal subject of an app, it is unavoidable to have the interactions between components. The component, being accessed, provides a kind of resources,

which may contain sensitive private data like a content provider holding user's calendar. Such access is not limited only to content providers, but every component can potentially provide sensitive information. Android permission model allows a component to define permissions to protect itself from being arbitrarily accessed, and such accesses are primarily protected by permissions.

- A component can access resources without interacting with components. Such resources can be raw files, databases or io devices owned by system or other apps. The system does provide APIs to access resources directly. However, the component may bypass the legitimate APIs and access through other APIs, even the primitive system calls. Such resources are also protected by permissions but these permissions are backed by Linux kernel sandboxes.

1.0.5 Thesis Statement

In this thesis, we developed a thoroughly layer-based mandatory access control solution to confine the privilege of Android Component. We considered Android component as the principal subject and built the access control to protect resources discussed above. To prevent unwanted access, we developed Flasa, compac and TzNFC to contain the power of a component. Some other work may focus on the isolation approach at different levels to separate the context, while we focus on building the controlled access in the system to maintain one device user experience, providing flexibility to the users. As a multi-layer solution, our work impacts multiple system layers including Android framework, database engine, Android runtime, Linux kernel and TrustZone.

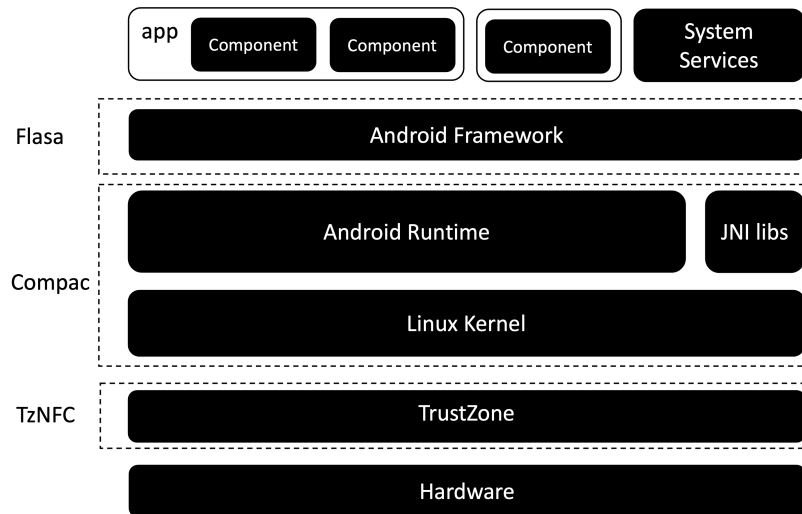


Fig. 1.2.: Solution Stack Overview

At framework layer, we developed Flasa to control the intra-component invocations and database accesses. to control a component's access to others, A intercept layer is built in the process lifecycle including the start process (Zygote), Intent IPC, broadcast mechanism, and content provider. To be able to segregate data generated by different component, a data isolation mechanism is built inside SQLite database engine to enable automatic data labeling.

Compac, further more, developed reference monitors and isolation at Android runtime and Linux kernel. We extend the Android permission mechanism to component level, and enable a component to have its permission set. At Android runtime layer, Dalvik as a typical Java virtual machine is modified. A set of hooks is inserted into Dalvik, making it record the dynamic call chains of Java class so that a component's dynamically running code is intercepted when claiming a permission. Some privileged activities relies on the privilege of Linux native kernel. Therefore, at kernel level, a kernel reference monitor can intercept the permission request of native privileged activities and traced back in Dalvik

for the current running component, so as to make the jurisdiction of whether allowing such access.

As a comprehensive solution, we realize some resources (such as Host card emulation service of digital wallets, user's PII) may require extra protection and only be accessible by given services under extreme condition like device is rooted or compromised. Take the digital wallet as an example, for device with TrustZone supported and enabled, we developed TzNfc, a solution to control a component's access to highly sensitive financial data including tokens and credit cards. TzNfc builds a Trusted computing base (TCB) inside secure world of TrustZone. A secure channel is established between the TCB and the resources to be protected. By doing so, a component as the access principal, only with the related privileges, can access the privileged resources.

2. RELATED WORK

Resource isolation is a key consideration when controlling access. Various of technologies have been employed to isolate resources or control the access of resources.

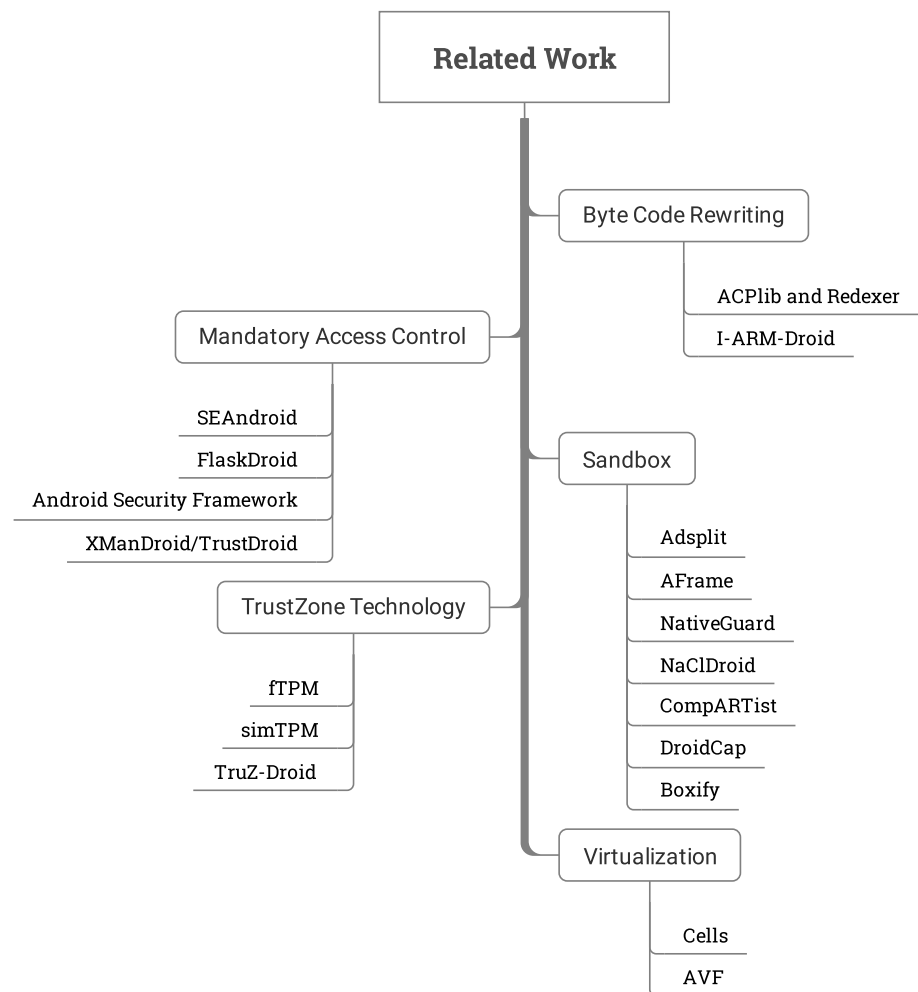


Fig. 2.1.: Related Work

2.0.1 Sandboxing

A collection of work [1–8] uses kernel sandboxing technology to create multiple processes for the purpose of isolation. AdDroid [1], AdSplit [2] separate Ads libs in the other process from the main app process, while NativeGuard [6] and NaClDroid [7] separate native code in a newly created process. Aframe separates UI elements in a standalone process from backend logic to isolate UI IO. DroidCap separates untrusted libraries in a standalone process and uses Binder handles, as capability token to delegate a restricted set of permissions to the untrusted libraries in order to confine, delegate or revoke their permissions. Boxify [5] allows untrusted code to run in an app level architecture called Boxify, Boxify dynamically load and execute the untrusted app in one of its own process and controls the binder IPC and syscall via its sandbox services.

2.0.2 Virtualization

Virtualization technology is widely used to isolate rich OS contexts and provides multiple OS instances experience to the users. Cells [9] leverages lightweight OS virtualization to isolate virtual phones from one another. It does not include a EL2 hypervisor for the rich OS instances, instead, the kernel is modified to behavior as a hypervisor to manage multiple virtual phones. Each virtual phone is assigned with unique virtual namespaces, device namespaces, identifiers and kernel interfaces but shares a single kernel. Compared to Cells, Android Virtualization Framework (AVF) [10] provides a more fundamental virtualization with an EL2 layer. AVF leverages pKVM as hypervisor to virtualize the whole OS including Linux kernel (EL1) and the above framework stacks

(EL0). AVF is integrated in Android open source project (AOSP) and can be enabled when compiling the system image.

2.0.3 ByteCode rewriting

I-ARM-Droid [11] and other work [12–14] builds reference monitors inside apps by rewriting the code to achieve a certain control of accessing critical resources. The benefit of such method is to remain the system and framework unchanged and the added control is enforced inside each app. However, such method is not generic to all apps and cannot be easily spread to all available apps.

2.0.4 TrustZone technology

A set of work [15–17] takes advantage of TrustZone technology to implement TPM on all kinds of platforms. fTPM [15] implement a software-only TPM Chip upon TrustZone OS and hardware. Thus, the whole TCB comprises the ARM SoC hardware, the TEE and the fTPM services. In particular, simTPM [18] is built with the base of fTPM and extend TPM to mobile devices.

TruZ-Droid [19] enables TrustZone in Android and builds an end-to-end secure communication between the remote server and Android device’s TrustZone-based TCB. Both TzNfc and TruZ-Droid build a secure channel in a certain way, but the two work has several differences. Firstly, TzNfc focus on fine-grained access control while TruZ-Droid focus on the end-to-end data protection. TruZ-Droid protects the communication channel with remote server and ensures the local data cannot be arbitrarily accessed by

unauthorized app, while TzNfc builds fine-grained access control by leveraging a secure channel between the subject component and resources. Secondly, TzNfc makes an integration with native hardware compared to solutions like TruZ-Droid. TruZ-Droid enables TrustZone with external I/O on HiKey board, while TzNfc integrates the solution with native Samsung Soc hardware and NFC controller.

Similar to TrustZone, a batch of work [20–22] work with Intel SGX fulfill a similar EL3 TCB responsibility towards data protection.

2.0.5 Mandatory Access Control

SEAndroid [23] embeds SELinux and implements domain type enforcement in Android. As part of stock Android OS, SEAndroid supports various resources protection at kernel level. Our work is SELinux policy compatible, but focus on framework level IPC and user resources' protection, which complements SEAndroid's MAC.

XManDroid [24] and TrustDroid [25] define their own policy and provide a framework level Flexible Mandatory Access Control on Android. They enforce isolated domains on ICC, the file system or Internet socket connections by leveraging the TCG Mobile Trusted Module, TOMOYO Linux and a modification of Android framework. FlaskDroid [26] extends the policy to Linux kernel, which is similar to SEAndroid but with different implementation and domain type enforcement.

Android Intent Firewall(AIF) and Android Security Framework (ASF) [27] builds MAC in Android framework. AIF builds a firewall upon the intent mechanism (the most popular interaction method) and controls the component interaction through intents. ASF prots

AppOps and AIF as a module and enable the framework level component interactions.

Inside the kernel, Linux Security Module (LSM) and the BSD MAC framework are employed, and FlaskDroid was introduced to support framework security context. ASF, as well as AIF, achieves a similar goal of the ICC control part of Flasa, but there are some differences. Firstly, the implementation is different. Flasa is a type enforcement extension of SEAndroid in framework while AIF builds without kernel MAC and ASF builds on BSD MAC framework. Secondly, Flasa focus on all scenarios of ICC and SQLite data control. Intent-based ICC is one of the most popular scenario. Thirdly, Flasa builds component interactions from scratch while ASF ports AIF. In addition, the two work are built on later Android version (4.4.2 and 4.3) while Flasa was built on 4.0.4.

3. A General Security Architecture for Diverse Mandatory Access Control Policies on Android

3.1 Introduction

Security has become a significant issue with increasing malicious and vulnerable mobile applications on Android. Current Android security relies on two levels of access control: capability-based permission control, which is enforced in Android framework, and access control list (ACL)-based sandboxing mechanism, which is enforced by Linux kernel. Both levels of access control take discretionary access control (DAC) approach. It has been widely studied that DAC cannot support certain security policies such as strong application and data isolation, and cannot control inter-application communications with flexible security policies [28].

Several security extensions have been proposed in Android framework to enforce constraints on an application's declared and runtime permissions [29–33]. However, these proposals do not solve the problems since they still rely on DAC-based Android permission model. Mandatory access control (MAC) has several advantages beyond DAC, such as strong isolation among domains (e.g., processes, data, and other resources), and information flow control [28]. MAC features have been proposed in Android to confine the runtime permissions of an application [24, 34–36].

However, each of these prior arts focuses on a particular purpose and enforces a particular MAC policy, while lacks a general security architecture for loadable security policies. Furthermore, these approaches do not distinguish runtime privileges of individual components rather than applications. For example, all existing solutions cannot control inter-component interactions at Android component level, cannot prevent malicious applications' intercepting broadcast intents [37], and cannot enforce fine-grained access control from different applications to individual data items of a content provider [38] (See Section 3.3.2 for more analysis). SEAndroid [23] is a systematic approach that employs SELinux on Android to enhance the security foundation of the whole platform, e.g., to confine vulnerable system daemons and control the interactions of applications in Linux kernel. In application framework level, SEAndroid supports several middleware MAC mechanisms, such as install-time MAC, permission revocation, intent MAC, and content provider MAC. However, current SEAndroid does not provide fine-grained and systematic inter-component access control in Android application framework. For example, current SEAndroid does not support component-level labeling and label transition, and a content provider cannot distinguish data items with different security labels.

In this work, we argue that MAC in application component level is essential for Android. First of all, with very flexible component sharing and re-using mechanism provided by Android, a single application can have multiple entries, each can be invoked by different applications with different security contexts (e.g., sources and groups of the applications). However current Android framework does not distinguish the privileges of individual components of a single application during runtime, which we believe is the root cause of many attacks in Android such as intercepted broadcast [39] and unauthorized data

access through content providers [38]. Secondly, Android framework does not dynamically change the security context of a component within an application when it is invoked or accessed by others, e.g., a less privileged application. This easily results in permission escalations [40] and over-privileged pre-installed applications [41]. Last but not least, current Android lacks a strong isolation among components and data during runtime, which is one of the main properties for many scenarios where information flow should be controlled, such as in Bring-Your-Own-Device (BYOD) for enterprises.

In this work, we design and develop Flux Advanced Security Android (Flasa), a general Flask-based security architecture in Android framework layer. Following the design principal of Flask architecture [28], Flasa takes a clear separation of policy specification and enforcement mechanism:

- Toward flexible MAC policy specification, Flasa adopts an Android-specific access control model where subjects are active components such as activities and services, and objects are passive components and data objects served by content providers. With different inter-component communication (ICC) methods among components during runtime, we define 9 access actions between a subject and an object. With these primitives, label-based security policies can be defined to achieve different security objectives, such as domain isolation and transition, and information flow control.
- Toward a general security architecture, we insert security hooks and security server in Android application framework, so that all applications extending basic components in Android are controlled by Flasa automatically. Particularly, we extend the binder

in Android to assign unique component identifier (CID) and a security label to each component when it is invoked. The mapping of CIDs to security labels are recorded by a system table maintained by the activity manager service (AMS) in Android system process. Upon an access request between a subject and an object, the AMS intercepts the request through ICC and queries a label management service (LMS), which is the security server of Flasa. The LMS in turn queries loaded security policies and returns an access control decision. The system process that holds the AMS and the LMS can dynamically load policies with APIs provided by Flasa. Optionally, each application can load policy or include policy rules in its `AndroidManifest.xml` file, but can only have label specifications and policy rules for applications signed by the same key.

We have implemented Flasa in SEAndroid ICS 4.0.4. Since Flasa does not hook individual applications, it does not introduce extra burden to application developers. In our implementation, Flasa leverages SEAndroid policy syntax and `libselinux` APIs to manage and access policies. Our implementation has 2152 LOC on ICS 4.0.4. By extensive evaluation with existing malware and our developed applications, we demonstrate that Flasa achieves its design objectives with very efficient policy definition. Our performance evaluation shows that Flasa introduces trivial overhead.

The work is organized as follows. We introduce necessary background of Android in Section 3.2. We analyze the problem and motivation in Section 3.3. We give an overview of Flasa in Section 5.3, and then elaborate our access control model and enforcement architecture in Section 3.5 and 3.6, respectively. We discuss some limitations of Flasa in

Section 3.8 and present our evaluation results in Section 4.5. Section 5.6 presents related work and Section 3.11 concludes this work.

3.2 Background

Android Application Model. An Android application can have one or more components, each of which is one of the four types [42]: activity, service, content provider, and broadcast receiver. An activity provides a UI for users to interact with. A service works at background to offer computing functionalities, e.g., for heavy or long-term tasks. A content provider holds data that can be shared with other applications. If an application wants to access data in a content provider, it has to use standard APIs offered by Android framework. A broadcast receiver registers for broadcasts sent out by privileged applications or system services.

Each component can be an entry-point for an application, therefore an application can have multiple entry points, which is very different from traditional Linux program. Android application is also event-driven. The framework uses callback functions to capture user or component events. Both user-triggered events and inter-component interactions boil down to inter-component communications (ICC) in Android, e.g., invoking activities, starting/binding services, opening content provider, or sending broadcasts. All these events are captured by the activity manager service (AMS).

System Services. Android has a set of services that provide system resources and support for applications [43], such as the activity manager service (AMS), package manager

service (PMS), and battery service. All these services reside in a process called `system_server` with `system` UID, therefore it runs with the system privilege.

In Android, a component cannot directly get the reference of another component, therefore any inter-component interaction such as `invoke` or `query` falls into one of the two types of ICC mechanisms: `intents` and remote procedure calls (RPCs). `Intent` is asynchronous ICC, and each `intent` is delivered to the AMS, which analyzes and routes it to a target component. RPC is synchronous and looks like a local method call. For example, an activity can issue an RPC to query a content provider, and waits for the RPC to return before moves forward.

In Android, the AMS intercepts all asynchronous ICC and some synchronous ICC like opening content providers. The PMS maintains all applications' package information including UIDs and permissions, and it helps the AMS make decisions when an application's permission is checked. Therefore the AMS and the PMS jointly act as the reference monitor of the Android permission model.

Android Security Architecture. Android puts access control in both Linux kernel and Android application framework [44]. At the kernel level, Android uses Linux process isolation to protect system resources. Besides, Android makes two modifications: first, Android implements sandboxing by assigning each application a unique UID, so that one process cannot arbitrarily access the resources of another process. Second, Android enforces access control to privileged system resources (e.g., Internet access) with Linux kernel group IDs [36]. At the framework level, Android develops its permission model to restrict applications from accessing protected APIs, e.g., using camera functions and

location data. Android also assigns a permission for each protected resource. For example, if an application wants to read SMS messages, it has to declare the `READ_SMS` permission in its `AndroidManifest.xml` [45] file.

3.3 Problem Statement

In this section, we discuss the problems that Android permission model, we also discuss the previous work to see the limitation. We also point out the challenges that we are facing to design a new protection model to compensate the weakness of Android permission model.

3.3.1 Defects of the Android Permission Model

Android enforces capability-based access control to system resources by defining a permission for each resource. Each application declares its required permissions in its manifest file, which are authorized by users during installation. However, Android permissions are checked based on UID – all components of an application have the same set of permissions, and there is no runtime constraints or downgrading of these permissions. This results in several security vulnerabilities. We highlight three in this work which have been identified in the literature and then analyze their root causes. We do not believe this is an exhaustive list.

However, it fails to protect the system due to Android’s new features. The event-driven feature causes frequent android component interactions between applications. The event-driven feature allows an android application has multiple entry points. each activity

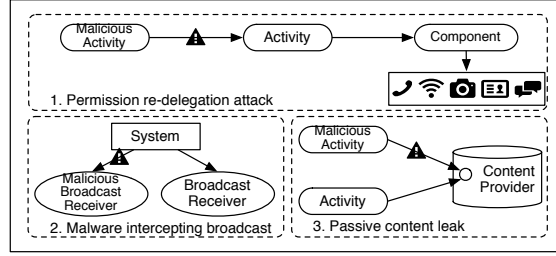


Fig. 3.1.: Defects of the Android permission model.

or service can be invoked directly by user's click or another application. These features enhance use experience, meanwhile, introduce vulnerabilities.

In general, the vulnerabilities can be concluded into three folder as depicted in Figure 3.1.

No permission restriction along activity call stack. Android does not enforce restrictions between activities; therefore, an activity without a particular permission can leverage another activity with the permission to access privileged resources, as shown in Figure 3.1(1). Many malware leverage this property to launch attacks, such as privilege escalation [35], permission re-delegation [35] and component hijacking [46] attacks.

No mandatory control on broadcast receivers. Any application can receive Android's broadcasts once it has related permissions, e.g., `RECEIVE_SMS`. However, the system cannot distinguish applications that have such permissions. Therefore, as shown in Figure 3.1(2), a malicious application can promote its priority to receive a broadcast before any benign application. After having received the message, it asks the system to abort the broadcasting message so that benign applications cannot receive it [39].

Data leakage and unauthorized access to data in content providers. In Android, each content provider can serve multiple applications, and can define a permission to

protect its data. However, the permission on accessing content providers is too coarse such that it cannot distinguish a caller application's privileges at data item level. This results in two issues. First, untrusted applications may read trusted data without user's consent or authorization, as shown in Figure 3.1(3), which results in passive content leak [38]. Second, as long as a caller application has the permission to read the content provider, it has privilege to modify or even delete the data written by other applications. This is an implicit permission escalation in content provider.

source applications may belong to different domains, and usually have different privileges.

While their data are stored by the same content provider, data isolation is usually desired, e.g., an enterprise application's data should not be altered by an untrusted application. Current individual data items in behind databases. This leads Furthermore, have different trustworthiness. For example, an application developed by top developer from Google Play and a third-party application from less trusted resources may have different trustworthy, so they may have different privileges on writing/reading data. Since current Android permission model cannot distinguish the caller application's privilege at data item level.

3.3.2 MAC for Android

It has been widely argued that capability-based DAC lacks dynamic permission revocation, controlled invocation path, and strong domain isolation [28,47]. This motivates MAC policies and enforcement mechanisms in Android framework.

To summarize the discussion above, we realize that Android need a two-dimensional protection depicted in Figure 3.2. Only the vertical protection is not enough for such a multiple-entry point, high inter-component communication system. It also needs the horizontal protection between applications.

Besides, For the content provider, its content provider is too coarse, the data from different applications with different privileges. However, by storing them into the same content provider, the privileges of data get the same privilege.

Several MAC mechanisms have been proposed for Android. SEAndroid [23] brings SELinux kernel into Android, with the focus of protecting processes and resources, such as system daemons, hardware, and private data at the kernel level. At the Android framework level, SEAndroid implements a simple install-time MAC policy [23]: mandatory control of applications' permissions so as to restrict the privileged resources. Therefore, SEAndroid does not provide general security architecture for flexible MAC policies at the framework level, as it does in the kernel, and its MAC policy cannot solve the aforementioned defects.

To solve the permission escalation problem caused by component interactions, several solutions have been proposed, including XMandroid [24], TrustDroid [36], Quire [34], and IPC inspection [35]. However, these solutions have several limitations and they cannot solve other problems. First of all, the policies enforced in these solutions are usually too strict: they all try to use the intersection of the permissions of applications along a call stack. Second, their protections are coarse-grained: they essentially enforce control on application-level interactions, which inherits the limitation of current Android permission model, e.g., uncontrolled data access through content providers, and no MAC protection for broadcast receivers and services.

The defects of current permission model and the shortcomings of existing MAC-based approaches motivate a *general security architecture at Android framework level* to control interactions in the *granularity of components*, toward enforcing *diverse and loadable MAC policies*. Toward this objective, we design and implement Flasa – Flux Advanced Security Android, a flask-based security architecture in Android framework, which complements SEAndroid. We illustrate the access control model, architecture, and implementation of Flasa in the rest of this work.

The horizontal protection has to be fine-grained at component-level, flexible enough for user to use. What’s more, the horizontal protection needs to cover all the inter-component interactions while avoiding overprotection.

For all the existing solutions XAndroid [1], quire [2], IPC inspection [3]. They also leverage Android’s permissions as authorization token. However, permissions are designed for fine-grained system and user resources control. To move the permissions for component-level protections, it will confuse the concepts of component and privileged resources. We hope to design a new authorization token, which has a closed privilege for component instead of resources.

In this case, the new horizontal protection model and the vertical protection model – Android permisisions – collaborate with each other.

The new model also should be easy to configure the policy. As a advanced protection model, users can pick a security level for their roles or easily turn off the protection based on their roles. The model can protect the users without frequently modified. In this case, the new model will sacrifice little user experience while improving the security a lot.

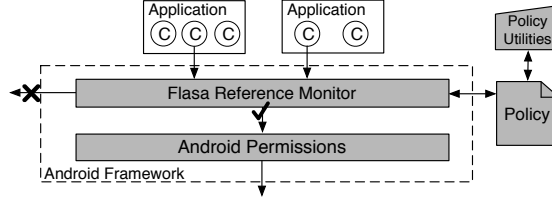


Fig. 3.2.: Flasa overview. Flasa enforces MAC policies orthogonal to the Android permission model.

The new model should also be expendable. For example, it can easily be extended to protect the web applications, or for the BYOD purpose.

3.4 Overview of Flasa

We give a high-level picture in Figure 3.2. Flasa is a MAC layer on top of the existing Android permission model, consisting of a reference monitor in Android framework and MAC policies. Specifically, when there is an ICC triggered by a component in an application, the Flasa reference monitor intercepts the event, and obtains the security contexts of the requesting component and target component or data (e.g., a data item in a content provider). It then queries loaded security policies with such information, and decides whether the access should be allowed or not. Once allowed, the execution goes as usual – the Android permission model further decides if the access should be allowed or denied, based on the permissions declared by the application. In this way, Flasa complements the Android permission model in a seamless manner.

Flasa adopts a label-based MAC model for policy definition, where a label defines a component’s security context. A component’s initial label is inherited from its application package. When invoked, its label can be dynamically transited to a new one, which

depends on the accessing component’s label and the type of access action, specified by MAC policies. Flasa also maintains the label of each data item of content providers. When a caller component writes new data into a content provider, Flasa assigns the caller’s label to the data item; when a caller component reads data from the content provider, it only returns the data that the caller can read by querying related policies.

Design Strategies. Flasa follows several design principles of Flask architecture, such as loadable security policies, clean separation of policy and enforcement, and security decision caching. Particularly for Android framework, Flasa takes several unique design strategies.

First, Flasa enforces security policies at Android framework level, therefore it inserts security hooks in the framework only. We do not hook individual applications, which usually requires application developer’s involvement. For example, a service in Android runs in background and provides data and functions to other applications. However, there is no distinct boundary for the service and its provided data and functions, which are usually proprietary logic of its developer. Therefore in Flasa, we do not enforce access control to the data and functions provided by the service; instead, we focus on the privileges of binding the service component.

Different from service component, all data items maintained by a content provider are stored in an SQLite [49] database file, and Android framework provides standard APIs for the content provider to access the data items, e.g., for `query`, `create`, `insert`, and `update` data in the database file. Similarly, Android framework also provides standard APIs for an application to access the content provider with similar functions. Therefore, Flasa labels

individual data items in content providers and enforces security policies in very fine granularity, by inserting security hooks in the base class of the content provider.

Similar to SELinux, Flasa provides mechanism for device administrators to remotely load security policies, e.g., for enterprise IT administrators in BYOD. Very different from the loadable policy module in SELinux, many 3rd-party Android application developers have no motivation to write security policies. Therefore the design of Flasa provides strong isolation among applications and data with simple policy. For security-incentive developers, Flasa provides mechanism to embed policies into application package, which are loaded either at installation or update time. Most importantly, policy makers specify security labels only at application package level, but not in individual component level. During runtime, a component's label (security context) is either inherited from its application package or derived from policy rules when it is invoked by another component. This makes security policy small and easy-to-write.

3.5 Access Control Model

3.5.1 Subjects and Objects

Flasa enforces access controls by intercepting ICC and data access, which means that a subject is a component, and an object can be a component or a data item in a content provider. In Android, the class `android.content.Context` provides connection to Android system and other applications [50], therefore any component that has the context can interact with other components. Only activities, services and applications can actively interact with other components, since they extend the `Context` class. With an activity,

Table 3.1: Access modes in Flasa

IPC Types	Modes	Subject	Object
asynchronous message	invoke	all ¹	activity
	start/stop	all	service
	bind/unbind	activity	service
	send	all	broadcast receiver
synchronous call (RPC)	open	all	content provider
	read	all	data
	write	all	data

¹ all means activity, service or application.

service, or application's context, a content provider or broadcast receiver can interact with other components. In other words, content providers and broadcast receivers cannot be subjects.

3.5.2 Access Modes

Access modes are the actions that a subject component takes to an object component or data. To generalize the types of ICC, we define 9 access modes as $\mathcal{M} = \{invoke, start, stop, bind, unbind, send, open, read, write\}$, with the semantics depicted in Table 3.1.

Access modes **invoke**, **start/stop**, **bind/unbind**, **send**, and **open** are performed through asynchronous **Intent** messages. Any component can **invoke** an activity in order to perform a new realtime task on the screen. A component can **start** or **stop** a service. However, if the component wants to invoke a remote procedure call (RPC) to the service, it has to **bind** the service. After the component finishes the task, it needs to **unbind** the service. In Android, only an activity can **bind/unbind** a service. Besides, a component can

send a broadcast to broadcast receivers and it can **open** a content provider. Once it can open a content provider, it has the privilege to create/insert data.

Access modes **read** and **write** are triggered through synchronous calls, which can be defined by Android Interface Definition Language (AIDL). Mode **read** allows a component to query data, and **write** allows the component to update or delete data in a content provider.

3.5.3 Labels

Label is the only security context to identify a component's privilege in Flasa. We define L as the set of labels. L_S and L_O are the sets of subject labels and object labels, respectively. Each application, component, or shared data item has an associated label. Flasa is a centralized label model, where a label is a string token, and the mapping from component identifiers to labels is maintained by the **system_server** process. None of normal privileged process can modify this mapping.

A component's initial label is inherited from the application package that it belongs to, which can be defined in two possible ways. First, an application can claim its label in its **AndroidManifest.xml** file. In order to ensure the uniqueness of the label, the label can have the property of **shared-label="no"** if it does not need interaction with others.

Second, a device administrator can define application labels in a security context file, e.g., loaded through a web service interface provided by enterprise IT. A context file can define multiple applications' labels with their certificates or package names. In case an application's label is defined in both a local context file and its manifest, the local context

definition overwrites the manifest, in order to ensure the mandatory control of the device's security policies. Note that when there is no label definition for an application, Flasa defines a default label for it. A sample context file is shown as follows:

```
<signer signature="@PLATFORM" >
    <label name="platform" />
</signer>
<package name="@DEFAULT" >
    <label name="default" />
</package>
<signer signature="308203ff308202e7a003020102020900b8...a4cf">
    <label name="comp.A"/>
</signer>
<package name="com.android.launcher">
    <label name="launcher"/>
</package>
```

When launched by users, an activity is labeled with its initial label. When invoked by another activity, its label is decided by available security policies. After the component is labeled, all ICCs are controlled by Flasa according to policies. During runtime, components with the same label compose a logical domain, which can be in one application or across different applications.

3.5.4 Policy

In general, we define the set of policies \mathbb{P} as $\mathbb{P} = ACP \cup LTP$, where ACP is the set of access control policies (or policy rules), each of which is a triple $(l_s, l_o, modes)$, where $l_s \in L_S$, $l_o \in L_O$, and $modes \subseteq \mathcal{M}$. This indicates that subject label l_s is allowed to access an object label l_o with access $modes$. For example, the following rule states that subjects

labeled with `comp.A` can perform the access modes of `invoke`, `open`, `read`, and `write` on target component labeled with `default`. For convenience, Flasa also supports symbol `*` to stand for any label.

```
allow comp.A default:component {invoke, open, read, write}
```

LTP is the set of label-transition policies, each of which is defined as a tuple (l_s, l_o) , specifying that when an object with label l_o is accessed by a subject with label l_s with mode `invoke`, its label has to be transited to l_s .

For example, the following transition rule states that an object activity's label is changed from `comp.A` to `default` if the access request from `default` to `comp.A` is granted.

```
allow-trans default comp.A:activity
```

Given a subject label l_s and an object label l_o , the label checking procedure outputs two results: grant or deny the access request, and optionally the new object label l'_o . We define the access control procedure as a quintuple $\mathbb{A} = (L_S, L_O, \mathcal{M}, \Sigma, F)$, where L_S is the subject label set, L_O is the object label set, \mathcal{M} is the access mode set, $\Sigma = \{auth, create, trans\}$, F is the output and $F = \{ \{grant, deny\}, L_O \}$.

Every time when there is an access request, Flasa reference monitor takes a triple $(l_s, l_o, mode)$ and makes three types of decisions based on policies. First, it decides whether to grant the access request. This procedure $auth(l_s, l_o, mode)$ can be defined as $L_S \times L_O \times \mathcal{M} \rightarrow \{grant, deny\}$.

Then, when the access mode is `write`, which indicates that the access is to create/modify/delete a data item in a content provider. The label creation procedure $create(l_s, mode)$ can be defined as $\mathcal{M} \times L_S \rightarrow L_O$. For example, if a subject with l_s is

granted to *write* a data item, the data's label is assigned with l_s . If the data is modified instead of being created, its label will not be changed.

Lastly, the access control procedure decides how to assign the object's new label when the access mode is **invoke** and the access request is granted. The label transition function $trans(l_s, l_o, mode)$ can be defined as $\mathcal{M} \times auth \times L_S \times L_O \rightarrow L'_O$, where $L'_O = L_S \cup L_O$. For example, if a subject with l_s is allowed to *invoke* an object with l_o , the object label l_o is transferred to l_s as long as $(l_s, l_o) \in LTP$.

After the label checking procedure makes decisions according to the function set Σ , it returns the output F to the ICC handling service, upon which the service takes further processing for the operation.

3.6 Architecture

3.6.1 Architecture Overview

Figure 3.3 shows the enforcement architecture of Flasa in Android framework. Flasa can protect framework level resources including components and data. Component labeling and data labeling share the same design principle with different label modes. Component labels employ a centralized model and locate in the label manager service (LMS). Data labels utilize a decentralized model and are stored in their data containers. Both types of labels are carefully protected in label holders and the label holders are protected by either SELinux kernel or our add-on protection.

Whenever there is an ICC between components, either within an application or across applications, the subject component sends out an intent, which is routed to the activity

manager service (AMS) in the `system_server` process. The AMS captures the intent, extracts its access mode and the object component from the intent, and sends the triple $(l_s, l_o, mode)$ to LMS in the same system process. The LMS in turn looks up available access policies and checks if there is any matched rule. If there is a match, it follows the access control procedure to make several decisions, as explained in Section 3.5.4. The LMS then returns the results to the AMS, which include whether the access is allowed, and the object's new label. If allowed, the AMS sends the intent to the object component, by following the original ICC protocol. In case a new component object is created or invoked, the AMS asks the `zygote` to create a new process and then forwards the intent to it, e.g., to invoke a new activity or open a content provider.

If the object is an existing data item, AMS returns the readable/writable data labels to the object holder – a content provider, and the content provider allows the subject to read/write the data with the labels. When inserting a new data item, the content provider labels the new data item with the subject's label. If the access is denied, the AMS pops up a toast message to inform the user.

Reference Monitor. The reference monitor of Flasa is composed of two parts: an extension of the AMS, and the newly developed LMS, both residing in the `system_server` process. The reference monitor has system privilege and can effectively control all ICCs between components. We follow the three properties [51, 52] for the reference monitor design.

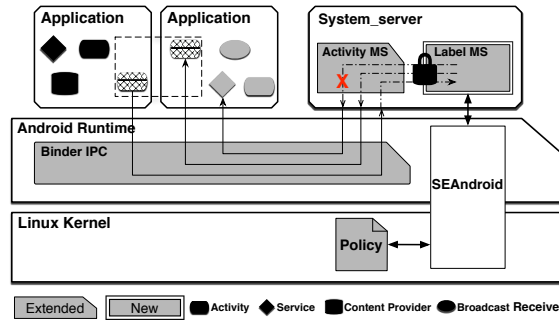


Fig. 3.3.: Flasa architecture. AMS and LMS consists of the reference monitor of Flasa.

- The reference monitor must always be invoked. Flasa uses the same reference monitor holder (`system_server`) as the Android permission model. For the access modes that Flasa supports, no ICC can bypass Flasa reference monitor.
- The reference monitor must be tamper-proof. We further employ SEAndroid to protect `system_server` process and Flasa's policy files, so that even if the device is rooted, the reference monitor, security labels and policies are still secure.
- The reference monitor must be small enough to be subject to analysis and tests, and the completeness of which can be assured. Our extension to AMS is small – 212 LOC changes and adds on ICS 4.0.4. The implementation of LMS is also very small – 500 LOC. We follow SELinux syntax for Flasa policy definition, which makes policy analysis and verification an easy task with existing tools from SELinux community [53].

Trusted Computing Base. The TCB of Flasa includes the Android framework, Dalvik and the underlying Linux kernel. It also includes the `system_server` process, which consists of the reference monitors of both Flasa and Android permission model.

3.6.2 Component Labeling

To successfully label a component, the first and most important challenge is to enable the AMS to recognize the component identity. Linux kernel provides traditional Linux contexts such as processes, sockets, and files, but cannot recognize any kind of Android contexts including components. However, Android needs component level context for inter-process communications (IPCs). Android uses binder to support this. We use a service binding operation as an example to illustrate how Android employs binder to achieve component context.

In Android, only an activity can bind a service. To send a service binding intent, the activity uses an RPC to the AMS called `bindService`. This binding RPC is defined with AIDL, and implemented by the binder. when the context switches from the application to the AMS, the caller-side library employs `bind` interface. We extend the binder implementation to pass the activity identity as subject to the AMS. The AMS receives the identity and searches the activity record hashmap for the activity¹.

3.6.3 Data Labeling

In Android, all data entries (e.g., rows of a table) in a content provider are stored in an SQLite database. We label each data entry by adding a new column called `FlasaLabel`. This column is exclusively controlled by the content provider and any user or application has no permission to modify it. A data label has the same semantics as a component label.

¹A component record is a data structure that the AMS holds to keep the component's information, such as its package name, UID, and process.

We put access control in the AMS for the `open` action, and access control in a content provider for `read/write` actions. When a component accesses the content provider, it first requests to `open` the content provider instance. Since only the AMS knows where the content provider is, the caller component has to send an RPC to the AMS to explicitly request the reference of the content provider. We put the `open` access control in the AMS to check if the caller has the privilege to open the content provider. In this case, Flasa's control is similar to related permissions in the Android permission model. Once the caller component obtains the reference of the content provider, it issues an RPC to read/write data from/to the content provider.

In Android, the caller component cannot access the SQLite database directly; instead, it can only access shared data through the APIs provided by the content provider, including `query`, `insert`, `update`, and `delete`. The content provider performs the database access actions on behalf of the caller. We put access control when the content provider handles such actions. Specifically, the caller can `query` data when its label can `read` the content provider's label, while the caller can `update` and `delete` data when its label can `write` data's label. The caller can `insert` data after it `opens` the content provider, and the created data entry inherits the label of the caller. Note that the caller application may need Android defined permissions in order to manipulate data, which is enforced by the Android permission model.

We add a new layer in content provider to make sure the AMS is involved in the data access procedure. We add an RPC in the AMS to allow the content provider to request readable/writable labels by sending the caller component's identity. As long as the content provider gets a remote request, it obtains labels from the AMS and attaches the labels into

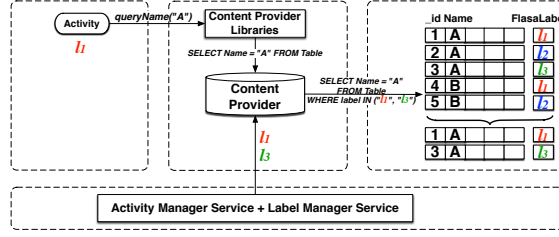


Fig. 3.4.: Data labeling and query. Activity with label l_1 can query data entries with label l_1 and l_3 in the database of the content provider.

SQL sentences. Below we explain how Flasa handles SQL queries and labels data entries when the caller component accesses the content provider with different operations.

Query. If a component wants to read data from a content provider, it issues a **query** operation. Suppose we have a sample policy rule as follows:

```
allow  $l_1$   $l_3$ :component {open, read, write}
```

As shown in Figure 3.4, when the component with label l_1 queries data, the content provider queries the AMS to obtain the readable labels, i.e., $\{l_1, l_3\}$ in this case, and appends "FlasaLabel in (' l_1 ', ' l_3 ')" to the WHERE clause of the SQL query sentence.

Insert. If a component adds new data into a content provider, it issues an **insert** operation. When the component with label l_1 tries to insert data, the content provider fetches the caller's label from the AMS with the caller's identity, and appends l_1 to the FlasaLabel column in the SQL sentence.

Update & Delete. The **update** and **delete** operations modify existing data. The difference between these two and **insert** is that these two operations allow the caller to modify a set of data entries by specifying a query condition. For **update** and **delete**, the caller's label should have **write** privilege to the data's labels, and the target data's labels do not change. When the caller wants to issue an **update** or **delete** operation, the content

provider asks for all writable labels that the caller can write from `system.server`, and attaches these into the `WHERE` clause of the SQL sentence.

The above four data operations achieve fine-grained data access control inside content providers by rewriting SQL sentences. Since the security checks are hooked in the base class of content provider in Android framework, applications are not aware of this, and there is no modification to application source code. During the whole procedure, only the Android framework knows the meanings of label and component contexts. The content provider merely enforces policy rules without interpreting access control semantics.

3.6.4 Policy Enforcement

Security policies are checked by the LMS in the `system.server` process, with added interfaces for the AMS to query access control decisions. The AMS converts component identities into labels and sends the triple (subject label, object label, access mode) to the LMS for making decision on an access request.

The LMS is the only place in the system that is able to store, load, and query Flasa policies. A content provider can obtain a copy of the policy rules that are related to its data labels and operations; however, no component in Android can modify labels and policies.

3.7 Implementation

We have implemented Flasa in SEAndroid 4.0.4, and deployed it on a Nexus S device. In total, we modify 19 files and add 2 files. The total added code has 2128 LOC. All these added code are independent modules, which can be patched into the Android Open Source

Table 3.2: Code modification statistics

Code File	File Number	Lines of Code	
		Modified	Added
Existing files	19	24	1624
Added files	2	n/a	504
Total	21	24	2128

Project (AOSP) automatically. We only manually modified 24 LOC in total. So it is easy to maintain the Flasa codebase. Table 3.2 summaries our code modification.

Switching component context. Binder recognizes the component identifier (CID) of an activity. We extend CID concept to all types of components. When a binder tries to switch application context from a subject to an object, it extracts the subject’s CID automatically and sends to the object through the binder’s driver in the kernel. After that, the object’s binder gets the request and the object is activated. In particular, when an `intent` is sent, the AMS get the CID and find the component’s record with the CID. We create a hashmap to hold the mapping from records to labels in the LMS. Since both the AMS and LMS reside in `system_server`, the AMS passes the subject record, object record, and access mode to the LMS without IPC overhead. After the LMS makes decision on the request, it returns results, and the AMS continues to handle the request.

Instrumenting content provider. We map out all SQLite operations (see Figure 3.5) for content provider and database access to make sure that there is no escape route for label attachment. For the compatibility between content provider’s access to SQLite database and an application’s access to its private database, we add the `FlasaLabel` column for all database tables. The private database only fills the column with a reserved label `sqlite`.

To attach labels, we use regular expressions to rewrite SQL sentences, and attach labels with different styles based on variant data operations. After rewriting SQL sentences, we add a validation layer in SQLite libraries to ensure that the modified SQL sentences have a valid format. For example, for the following SQL sentence,

```
SELECT _id FROM contacts WHERE _id NOT IN visible_contacts
```

if the table `visible-contacts` only has one column, it is valid; however since we add column `FlasaLabel` for all tables, SQLite throws a fatal error. By using regular expressions to validate SQL sentences, the SQLite libraries automatically rewrite above SQL sentence to the following:

```
SELECT _id FROM contacts WHERE _id NOT IN (SELECT _id from visible_contacts)
```

Leveraging SEAndroid. Flasa leverages SEAndroid in several aspects. First, Flasa’s policies have the same format as SEAndroid’s policies, which facilitates device administrators to configure and manage policies. Second, Flasa uses SEAndroid policy utilities to read policies, thus minimizes code modification. Third, Flasa relies on SEAndroid to protect critical processes and resources. System processes like `system_server` are protected by SEAndroid, and policies are only accessible by the `system_server` process. With these mechanisms, even if the device is rooted, Flasa labels and policies are still secure.

Since Flasa cannot label data in other types of data containers, like files and sockets, an application can intentionally open side channels (e.g., using local sockets or pipes) for others to access its private resources. To mitigate these threats, Flasa utilizes SEAndroid to control the interactions at Linux kernel level.

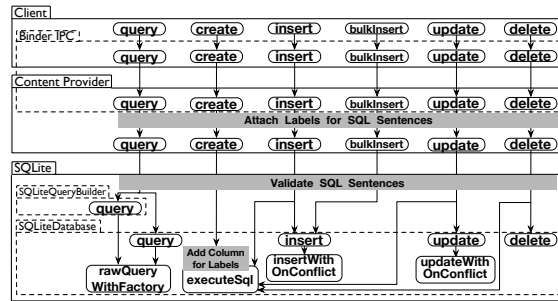


Fig. 3.5.: Content provider implementation details

3.7.1 Android Component Labelling

Activity Labeling. In Android, each activity instance has a record, which is stored by the AMS. An activity class can have multiple instances at the same time. Different callers can invoke different instances, while a user usually does not notice the difference among them. When the home button is pressed, the foreground activity is pushed into the activity history stack; when the back button is pressed, the foreground activity is destroyed but its state is stored for possible future use. Flasa labels different instances with different transition labels based on their callers. If a caller does not have the privilege to invoke an activity instance, based on security policies, the access request is silently denied, and the user still cannot tell the difference among different activities.

Figure 3.6 illustrates how the AMS intercepts ICC and enforces access control. When the AMS receives the `invoke` intent, it finds the target activity (callee) record and handles the related component states. After the AMS is ready to notify the target activity, it sends an asynchronous message to notify the caller component to pause itself. The caller process sends an asynchronous message² to its message queue [54]. Meanwhile, the AMS starts to

²This message is to pause its current activity so that the caller does not hold the screen.

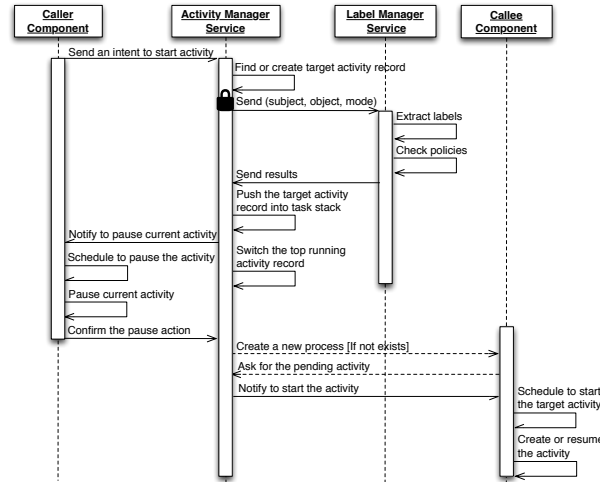


Fig. 3.6.: Activity invocation sequence. AMS enforces security policies when it identifies the subject and object activities.

switch the top running activity record. After the AMS receives the pause confirmation from the caller, it sends notification to the callee to start the activity. If the callee process does not exist, the AMS asks the **zygote** to create one. The callee process then puts the task – starting the activity – into its own message queue. Eventually, the callee’s activity is invoked and runs on the screen. Similar procedures are taken for starting/binding services or sending broadcasts.

We put inter-component control after the AMS recognizes the subject and the object, but before it takes further actions to switch processes. If the access request is denied by Flasa, the system simply posts a toast message to inform the user without crashing the application.

Service Labeling. A service request (**start/bind**) between a component and a service is sent to the AMS, so that the AMS can find the correct service. The AMS maintains a service record for each service instance, and each service class can only create one instance. Since a service shares computing resources to other applications on behalf of its own

application, once started, it is labeled with the application’s label, which is not changed at the runtime.

If the request is granted, the AMS starts the service and returns its reference to the subject component; otherwise, the AMS just notifies the user a denied message for the access mode `start` and returns a null pointer for the access mode `bind`. The null pointer causes the application to throw a null pointer exception. If the exception is not caught, the application crashes. This is similar to the scenario that the application has no Android permission to bind the service. Usually, the application developer catches exceptions thrown in RPC, so the deny decision does not introduce many user-experience issues.

Broadcast Receiver Labeling. Flasa handles broadcast receiver control slightly different from activities and services. Each broadcast has a record with the AMS. Flasa records all broadcast receivers in the broadcast record. The AMS intercepts all the explicit, implicit, and pending broadcasts. When the AMS captures a broadcast, it inspects each broadcast receiver’s label, and looks up matching policies with (subject label, broadcast receiver label, `send`). If no matched rule is found, this broadcast receiver is silently filtered out from the receiver list without notice.

3.8 Limitations

Components with different labels in the same application may escalate privileges by leveraging each other. Android makes lots of efforts on component reuse. Usually, one component cannot directly invoke another one even if they are within the same application. However, it fails to restrict the Java language features for sharing global public variables

and object references. Consider an application that has one activity labeled differently from the others, e.g., it is invoked by a less privileged component. This activity can read and modify a global public variable without control in the same Java application. The solution is a trade-off between usability and security. For application developers, we recommend they use more Android APIs and less global variables. For policy makers, if they want to use Flasa to effectively contain a potential malicious application, they can assign no label-transition rule for the application's activities. Two approaches can be taken to fundamentally solve the data and reference leakage problem: Linux kernel provides isolation for different components, or Dalvik provides a strong built-in Java language level access control.

Binder is used as part of the trusted computing base (TCB) for both Android permission model and Flasa. However, it is not absolutely trusted as it appears to. As part of the system libraries, an application's binder libraries are shared with the **Zygote** process, which forks all the main processes of applications in Android. These libraries can be reloaded by applications in Dalvik without control [55]. Many game applications use this mechanism to dynamically load their own libraries. To secure TCB, we restrict Dalvik from reloading binder libraries while retaining its ability to reload other libraries, for compatibility of existing applications.

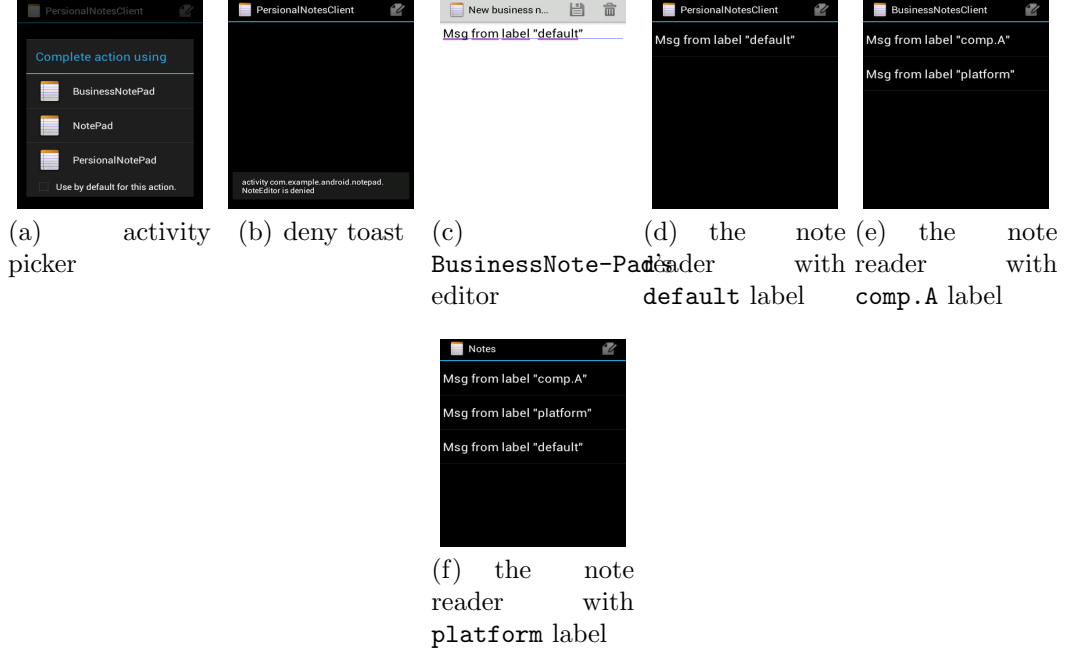


Fig. 3.7.: BYOD case study. Flasa enables a personal application to invoke the activity of an enterprise application. By transiting the label of the privileged component, this does not leak any sensitive data.

3.9 Evaluation

We conduct a BYOD case study and two malware case studies to demonstrate the applicability and effectiveness of Flasa and the flexibility of policy definition. We then conduct performance study to show that Flasa's overhead is extremely small.

3.9.1 Enforcing BYOD Policies

Bring-Your-Own-Device (BYOD) allows enterprise employees to bring their own devices and connect to the enterprise network, thus demanding strong isolation between data and applications from the enterprise and 3rd-parties, e.g., the downloaded games and social applications. Most existing BYOD solutions enforce coarse-grained access control, e.g.,

simply denying specific functionalities like GPS and network, which lacks flexibility and fine-grained control. Flasa provides customized configurations so that employees can make rich personal use of their devices without compromising the enterprise data and applications.

Suppose an enterprise IT administrator loads the context file as follows:

```
<signer signature="@PLATFORM">
    <label name="platform" />
</signer>
<signer signature="308203ff308202e7a003020102020900b8...a4cf">
    <label name="comp.A" />
</signer>
<package name="@DEFAULT">
    <label name="default" />
</package>
<package name="com.company.A.email">
    <label name="comp.A" />
</package>
<package name="com.android.browser">
    <label name="web" />
</package>
```

The administrator also loads access control and label transition policy rules as follows:

```
allow comp.A platform:component {invoke, start, stop,
    bind, unbind, open, read, write, send}
allow platform comp.A:component {read}
allow platform default:component {read}
allow default comp.A:component {invoke}
allow-trans default comp.A:activity
allow default platform:component {open}
```

With the above policies, any application from unknown source cannot access company A's data, which has the label of `comp.A`. However, the non-company applications still work

properly even by utilizing the application from company A to finish certain tasks, e.g., writing notes for its own with company A's notepad editor.

We use the sample application `NotePad` from Android SDK [56] to demonstrate the aforementioned scenario. `NotePad` has a content provider to store all notes on the device. We sign it with the platform's key and label it as `platform`. We develop two applications called `BusinessNotePad` and `PersonalNotePad` based on `NotePad` to enable them to use `NotePad`'s content provider, where `BusinessNotePad` is company A's official application signed by company A, and `PersonalNotePad` is from unknown source labeled with `default`.

When an employee wants to write memos for personal purpose, she opens `PersonalNotePad`, and clicks `edit`. An activity picker (Figure 3.7(a)) pops up and asks her to pick one. In the policy, the administrator disables `NotePad`'s editor while allows to call `BusinessNotePad`, because `NotePad`'s editor has potential data leakage and the employee has not updated it yet. When the employee chooses `NotePad`'s editor, the system posts a deny message (see Figure 3.7(b)) and returns to the previous activity. Then, for some reason, the employee picks `BusinessNotePad`'s editor, e.g., which has a better UI. In the original Android, this is a potential permission re-delegation attack, because the employee is using `BusinessNotePad`'s privilege to write notes. However, in Flasa, the employee can safely use it and leave a message (see Figure 3.7(c)). This message is labeled with `default` instead of `comp.A`, according to the transition policy rule. After that, the employee leaves a message in `NotePad` and `BusinessNotePad` separately. She can read messages from the three applications' note readers and get the results as shown in Figures 3.7(d)-3.7(f). The results confirm that the data are isolated according to their

labels. By using the transition rule, `BusinessNotePad`'s label is transferred from `comp.A` to `default`, which enables `PersonalNotePad` to use `BusinessNotePad`'s editor to write personal data without permission escalation risk.

3.9.2 Preventing Malware

Flasa naturally blocks many malicious behaviors with a simple policy definition as shown in Section 3.9.1. In such a policy, any newly installed application is assigned with `default` label if there is no policy rule that matches its developer's public key, or its `AndroidManifest.xml` file does not have related rules. To show how Flasa alleviates possible damages, we use `jSMShider` and `HippoSMS`, which attack the system at Android framework layer.

```
V/FLASA ( 92): Deny: defaultsystem (platform) cannot send Intent { act=android.intent.action.PACKAGE_ADDED dat=package:com.antutu.ABenchMark flg=0x10000010 (has extras) } to j.SMShider (default)
V/FLASA ( 92): Deny: com.android.phone (platform) cannot send Intent { act=android.provider.Telephony.SMS_RECEIVED flg=0x10 (has extras) } to j.SMShider (default)
V/FLASA ( 92): Grant: com.android.phone (platform) sends Intent { act=android.provider.Telephony.SMS_RECEIVED flg=0x10 (has extras) } to com.android.nms (platform)
```

Fig. 3.8.: Preventing malware. Flasa silently denies the malicious actions of `jSMShider`.

jSMShider. Flasa can restrict broadcast receivers. Malicious applications may register multiple broadcast receivers and a user usually grants required permissions during installation. `jSMShider` [57] signs itself with AOSP platform key in order to gain the system privilege in custom system image to install the second stage payload. It can register multiple receivers to listen to broadcasts, and takes further actions like reading and processing incoming SMS messages. Flasa successfully intercepts the broadcasts shown in Figure 3.8 and `jSMShider` is not aware of this interception. In such a case, the system does

not trigger `jSMShider` to read incoming messages. Actually, even if `jSMShider` explicitly tries to read/modify SMS messages, it still gets failed.

```
V/FLASA ( 115): com.android.mms open mms-sms
V/FLASA ( 115): com.android.mms open sms
V/FLASA ( 115): com.android.phone open contacts
V/FLASA ( 115): com.ku6.android.videobrowser:remote open sms
V/FLASA ( 115): com.android.mms open mms
V/FLASA ( 218): delete: attach labels "default" in com.android.providers.telephon
y.SmsProvider
```

Fig. 3.9.: Preventing malware. Flasa silently denies deleting SMS message from **HippoSMS**.

HippoSMS. HippoSMS monitors incoming SMS messages [58]. If the device receives a message from numbers starting with “10”, it deletes the message from the SMS content provider so that other applications including the system SMS message application cannot receive it. As shown in Figure 3.9, Flasa intercepts the malicious behavior from HippoSMS. Based on the policy, HippoSMS has the access mode **open** to insert new messages with its label, but has no right to read/modify the messages marked with other labels including **platform**, so that the SMS messages are intact after HippoSMS issues a delete operation to the content provider.

3.9.3 Performance

We build images from AOSP and AOSP with Flasa, and run them on the same Nexus S device. We evaluate the system and database overhead under the same phone state including the same set of applications and configurations. We find that the system overhead introduced by Flasa is negligible, and the database overhead is also very small with respect to the implementation of data label enforcement.

System Overhead. To evaluate the system performance, we employ the well-known benchmark application **AnTuTu** [59]. We run **AnTuTu** 10 times on the AOSP and Flasa

images separately, and calculate the mean scores for each performance measurement. As Figure 4.10(a) shows, the overhead of all aspects except the database IO is negligible. The 2Dgraphics and 3Dgraphics performance of Flasa is even better than that of AOSP. We guess this is because of software error. To better understand the high overhead of database IO, we conduct a detailed database IO evaluation.

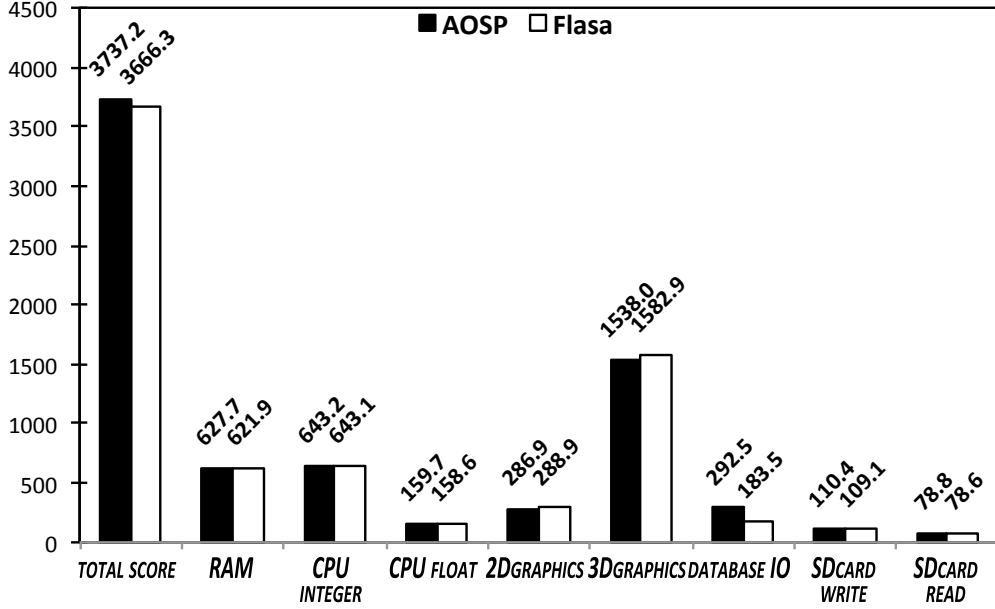


Fig. 3.10.: AnTuTu benchmark results.

Database IO Overhead. We use a database benchmark application called **RL Benchmark** [60] to investigate the performance of insert, query, and update operations. For a single IO operation test, Flasa sometimes performs even better than AOSP, so we simply ignore all single operation results.

The bundle operation results are shown in Table 3.3. We find out that the insert test in a transaction and select test contribute most of the high overhead. The access time is about 3-4 times on Flasa than that on AOSP. However, the pure insert operation does not introduce too much overhead. To make the SQLite database compatible with Flasa when

Table 3.3: RL benchmark results

	Operations	AOSP	Flasa
1	1000 INSERTs	62.411s	65.626s
2	25000 INSERTs in a transaction	7.238s	26.526s
3	25000 INSERTs w/ index in a transaction	7.446s	25.726s
4	5000 SELECTs w/ index	2.624s	7.523s
5	1000 UPDATES	7.318s	7.941s

attaching labels, we perform very strict regular expression checks to validate SQL sentences in `exeSQL` and query APIs. The regular expression is especially slow in Java because Java uses a backtracking implementation [61]. The insert transaction does not use `insert` API; instead, it uses the `exeSQL` API. This explains why the overhead of the insert transaction is much higher than that of the insert operations. In fact, most insert operations in Android applications are performed by `insert` API, because it uses parameterized statements to separate data and control, which avoids SQL injection attacks. Even Android team advises application developers not to use `exeSQL` API [62]. Therefore, the only overhead concern comes from the queries.

We believe that implementing the SQL validation process in SQLite’s native code without complex regular expressions will significantly reduce the high overhead caused by the Java regular expressions.

3.10 Related Work

A body of work provide install-time or runtime constraints for Android permissions [29, 31–33]. They extend the Android permission model by designing a new policy layer, which provides further fine-grained control on individual applications. However, these approaches usually enforce policies with the reference monitor of the

Android permission model. Moreover, they fail to control ICC, therefore cannot provide MAC protection in component granularity.

As aforementioned in Section 3.3, a body of pioneering MAC solutions have been proposed to address known privilege escalation and over privilege issues through MAC-featured component invocation control. TrustDroid [36] and XmanDroid [24] use a path-based TOMOYO Linux [63] as MAC kernel. Quire [34] and IPC inspection [35] only build MAC at framework without MAC kernel support. The difference between these and Flasa is that Flasa is a flask-based general security architecture in Android framework level, and has clear separation of policy specification and enforcement for very flexible MAC over component and data access.

Moreover, the SEAndroid [23] implements SELinux in Android Linux kernel and offers mandatory permission control for applications. SEAndroid tries to build as much middleware-oriented protection as possible and it has been merged into Android source code tree. As a milestone of Android security, SEAndroid has a set of middleware MAC extensions including install-time MAC, permission revocation, intent MAC and content provider MAC [64]. The install-time MAC is quite similar to [29, 32], and it applies an install-time check of application permissions. Permission revocation allows to revoke an application's permissions after the application is installed. Intent MAC provides control over the delivery of intents. Content provider MAC allows to control the functionalities of use, read and write content provider. Currently, only install-time MAC is merged into main SEAndroid branch and the rest three extensions are only available from each individual branches.

The most related MAC art is FlaskDroid [26,65], which provides fine-grained access control functionalities for OS vendors, application developers and phone users. By combining SEAndroid, FlaskDroid provides the most comprehensive protection for Android among all the previous work so far. FlaskDroid provides a flask-based solution and it supports a policy language that is quite similar to SELinux and the policy is enforced to Android components including activities, broadcasts, services, content providers. By providing APIs for application developer, FlaskDroid also allows application developers to have fine-grained control within services and content providers.

Our work and FlaskDroid have similar design goal and functionalities. Both of the work are context aware and try to provide a generic MAC solution for Android. FlaskDroid supports phone booth mode while our work supports BYOD feature. Both of them provide fine-grained control within content providers from different angle. However, the two work still have very big difference. The most difference between our work and FlaskDroid is the design principle. We try to design the MAC as a flexible and applicable solution from four aspects: first, we design Flasa with the smallest modification. We only modify/add 2152 LOC and can easily patch to the latest Android versions, which makes it is quite convenient for OS vendors to deploy Flasa. Second, we employ a simplified SELinux policy language in order to make the MAC management and usage much easier. Third, our fine-grained data protection is built in the system, while FlaskDroid needs the application developer to use its SDK to get the fine-grained protection for content providers because FlaskDroid provides an extension of the AIDL and its corresponding code generator.

Furthermore, Aurasium [66] builds an inline reference monitor in applications to implement fine-grained control. TaintDroid [67] and AppFence [68] taint data in Android

and detect information leakage when the data routes out of the context. ACG [69] provides a user-driven permission granting system, allowing users to grant permissions by click when a privileged resource is accessed. Permission Event Graphs [70] employs contextual policy by using static analysis and runtime monitoring to check the security properties of an application.

3.11 Conclusions

Flasa is a flask-based general architecture in Android framework to control inter-component communications and fine-grained data access control in content providers. Flasa enforces very flexible MAC policies. Uniquely, policy definition in Flasa is much simpler than that in traditional SELinux, e.g., its default setting provides strong data isolation among applications. Together with SEAndroid, we believe Flasa provides a solid foundation for complete MAC stack from Linux kernel to Android framework.

4. Compac: Enforce Component-Level Access Control in Android

4.1 Introduction

Android has become the most popular smartphone platform, taking more than 70 percent of the market shares [71]. Android uses permissions to restrict the behaviors of apps. An app (In the rest of the work, apps are used for Android applications) needs to have specific permissions in order to access protected resources. The app declares permissions that it needs in `AndroidManifest.xml`. During app installation, users are asked to approve the declared permissions. Upon approval, the app will be installed.

In Android, permissions are assigned at the app level. When an app is installed, it is assigned a unique UID, and each UID is associated with a set of permissions. At runtime, access control uses the UID to find out the permissions of an app, regardless of what component of the app is making the access. Therefore, all the components in the same app have exactly the same permissions. This is not a problem if all these components come from the same developer. However, this is not the case in Android and most other mobile systems. In these systems, apps often include third-party components. In-app advertisement is the most representative example. When an app needs to display ads, it has to incorporate the advertisement code (e.g. Google's AdMob). Once incorporated, both the ads and the original app will have the same privilege. Other commonly used third-party components include social networking service APIs, PhoneGap plug-ins, etc.

In most situations, apps need more permissions than what each component needs; when users grant the permissions to the apps, they also grant the same permissions to the components, leading to over-privileged components. In this case, some components in apps have more privilege than what they need. If they are malicious or have security flaws, they can cause problems. Previous work [72] indicates that several third-party components abuse apps' permissions to collect users' private information, without user consent.

Several ideas, such as AdDroid [1] and AdSplit [2], have been proposed to address the problem caused by a particular type of third-party component, namely, advertisement. AdSplit proposes to put ads in another process, isolating them from the app. AdDroid proposes to put ads into a service and assign a new `ADVERTISING` or `LOCATION_ADVERTISING` permission to this service. However, these solutions are mainly designed for advertisement; they are not general enough to extend to other types of third-party components.

In this work, we propose COMPAC (COMPonent Access Control), a more generic solution, which extends Android's UID-based permission model [44] into UID- and component-based permission model. More specifically, several untrusted Java packages can be grouped into components, and an app can be divided into different components; app developer assign permission sets for the app and its components in `AndroidManifest.xml`. In such a case, both the app and components' permission sets are subject to Android system's control. To the best of our knowledge, this is the first time to propose a generic solution to limit part of app code's privileges in Android.

We have implemented Compac in Android 4.0.4, and have conducted a comprehensive evaluation on it, not only on its performance, but more importantly, on how Compac can be used to solve the over-privilege problems faced by today's apps when they use

third-party components. Our results show that Compac is effective and applicable. We summarize the main contributions of this work as follows:

1. To restrict the privilege of third-party components, we propose a generic solution to contain in-app components' privileges by leveraging Java source code and extending the current Android permission model.
2. We implement the prototype called Compac and conduct comprehensive studies and performance evaluation to demonstrate that our approach can mitigate the damage caused by third-party code.
3. We provide app developers Compac APIs encapsulated in a customized SDK, which is compatible with the original Android SDK. We also develop a component permission manager to allow users to control app components' permissions on their devices.

4.2 Background: Access Control in Android

4.2.1 The “Windows” on the Sandbox

Fundamentally, Android uses Linux Kernel Sandboxing to isolate apps based on UIDs, however, merely having such a sandbox is too restrictive. Apps should be allowed to access the system resources, as well as each other's resources. However, such accesses should be controlled, not arbitrary. The sandboxing mechanism eliminates the arbitrary accesses. To allow controlled accesses, “windows” have to be opened on the sandbox, but behind each window, there should be an access control.

System Calls. System calls are a typical way to allow user-level programs to access kernel-level resources. Android uses system calls to access some of the protected resources. For example, accessing the Internet is done through system calls, i.e., only apps with `inet` GID can directly access these resources by making the corresponding system calls. When an app invokes a system call to create an `inet` socket, the system call checks if the app has the `inet` GID; if it has, the app can get the socket and be able to access the Internet.

Android Permissions. Behind the GID check in privileged system calls, there are Android permissions, making sure that the authorized non-privilege apps can access the intended resources. Take the `INTERNET` permission as an example. When an app with the `INTERNET` permission is installed, Android will assign the `inet` GID to the app and its processes, as one of its additional GIDs. When the app accesses the Internet, request will be granted because of the `inet` GID; Unlike kernel resources, Android framework resources including services, content providers and broadcasts cannot be accessed directly by the system calls. These system resources provide an Android IPC interfaces as the “windows” to normal apps and Android permissions access control is built behind each window. When an app with the required permissions tries to access the intended system resources. The system resources call framework reference monitor to check the caller’s permissions.

4.2.2 Reference Monitors

In the “windows” mentioned above, their access controls all have to get to one point: does the app have a particular permission? The windows themselves do not know the answer, they have to ask Android for its decision. In Android, this decision is only made in

two places: at the framework level or at the kernel. We call them Framework Reference Monitor (FRM) or Kernel Reference Monitor (KRM).

FRM resides in the `system_server` process and it consists of two system services: Activity Manager Service (AMS) and Package Manager Service (PMS). Activities, content providers, services, and broadcasts check permissions using FRM. Whenever they need permission check, they send an IPC to AMS, which works together with PMS to conduct the check. In some cases, the caller is already in the `system_server` process, so the call will be a local one, not an IPC.

KRM resides in the kernel. Conceptually, the Linux Discretionary Access Control (DAC) is considered as KRM. When conducting permission check, KRM cannot reach out to AMS and PMS for the app's permission information. Pushing the permission information into the kernel can solve the problem, but can introduce significant kernel-level modification. Android chooses to utilize the Linux DAC by leveraging the UID and GIDs of a process to make the access control decisions in the kernel. For example, accesses to bluetooth, sdcard and the Internet need GIDs of `net_bt`, `sdcard_r` and `inet` respectively.

4.3 The Compac Design

4.3.1 The Overview

The main objective of Compac is to provide component-level access control. In Compac, each app consists of one or more components¹. App developers or device users

¹Although Android platform has its own Android component definition, we use components to indicate Java packages. For example, Google Ads component means Google Ads related packages. Component is an abstract concept to facilitate the understanding of Compac.

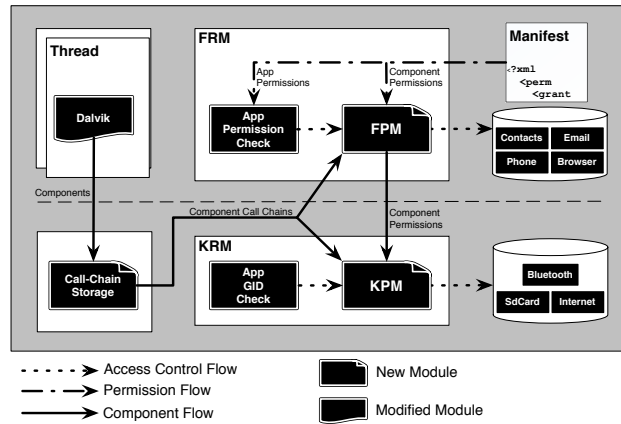


Fig. 4.1.: The Architecture of Compac

can grant different permissions to components. For example, an app has two components, one for the main activity, and the other for advertisements. The app can give only one permission (INTERNET) to the advertisement components, while assigning several permissions (such as READ_PHONE_STATE, ACCESS_COARSE_LOCATION) to the rest of the app. Such a fine-grained access control is not possible in the existing Android system: each app can have one set of permissions; once granted by the users, all the components in the app will have the same privilege.

The architecture of our design is depicted in Figure 4.1. It is composed of three pieces. First, an app developer defines the app’s permissions as well as the component permissions. When the app is installed, its app and component permissions are acquired by the reference monitors. As we mentioned in Section 5.2, Android has two permission-checking places: Framework Reference Monitor (FRM) and Kernel Reference Monitor (KRM). FRM still stores the app permissions as usual, and we do not modify the computing logic of the app permissions. To keep the best compatibility with current Android access control architecture, we extend the two RMs by adding two Policy Managers (PMs): Framework

Policy Manager (FPM) and Kernel Policy Manager (KPM). The two PMs hold the component permissions that are obtained from the app's `AndroidManifest.xml`.

Whenever permission update happens in Android framework, FPM synchronizes component permissions with KPM.

The second part is to extract the component information (Java package call chain) at runtime. We build hooks in Dalvik to trace the realtime Java method invocations. The trace is a call chain, which works like a call stack (FIFO). In order to make sure the component information is accurately recored, each thread of the app process has a call stack. If the thread is suspended, its call stack is locked. The thread's call stack follows the basic security rule. When a thread is forked from its parent, the parent's call stack status is inherited to avoid privilege escalation.

The third piece is access control enforcement. The RMs make decisions based on app's UID permissions and component permissions. RMs first check if the app has the requested permissions. If it does, RMs ask PMs to check the component permissions. PMs consider a call chain as the principal in an access control. They have the privilege to request the call chain from Dalvik. After the PMs finish the computation of permissions, they check the component policy to return the result to RMs; If the app doesn't have the requested permission, the request is denied without any further component permission check.

4.3.2 Assumption & Trusted Computing Base

Most Android apps are written in Java, but for performance reasons, Android allows apps to include native code (compiled from C/C++ code) [73]. To distinguish this type of

native code from that provided by the Android OS, we call it the app-specific native code. Since this native code runs in the same process space as Dalvik, if it is malicious, it can tamper with the process' stack and heap memory, including the memory used by Dalvik.

In Compac, Dalvik provides component call-stack information. If Dalvik's data memory can be changed by malicious components (through native code), there is no guarantee on the integrity of the runtime component information. In this work, because of the lack of isolation between the app-specific native code and Dalvik, we block the invocation of app-specific native code in apps. The assumption is only temporary and has limited impact:

1. Based on the previous study [74], only 4.75% of benign apps have native code.

Therefore, the majority of the apps will not be affected by this assumption.

2. Isolating the app-specific native code from the rest of the system is not impossible to implement. This goal is already achieved in the Chrome browser by the Native Client (NaCl) framework [75] using Software Fault Isolation (SFI) [76]. Robusta [77, 78] has successfully isolated the native code from Java Virtual Machine (JVM) in the traditional OSes. It will be just a matter of time before the isolation of the native code from Dalvik is achieved in Android.
3. If a component's permission set is the same as the app's permission set, i.e., there is no permission restriction on this component, we do allow the native code to be invoked by this component, because no extra privilege can be gained by this component even if it can modify Dalvik's memory. The blocking of native code is only enforced if a component has less permissions than the app.

It is quite tempting to enforce the component-level access control inside Dalvik, just like what the security manager does in the traditional JVM's security architecture [79]. In Android, the existing access control is not enforced inside Dalvik; instead, it is enforced either inside the kernel or in a privileged process (i.e., `system_server`). Android chooses to conduct access control in this way, rather than simply using the security architecture of JVM; this is mostly because Android itself uses a great deal of native code, in addition to the native code brought by apps. When an app invokes the native code, Dalvik will have no control. Thus we enforce the access control policy in Android framework not Dalvik. However, in order to secure component boundaries and ensure the correct runtime component information, we do implement some auxiliary security rules (see section 4.4.2) in Dalvik. In general, we only use Dalvik to get the runtime component information, not for access control.

4.3.3 Component Permission Configuration

In the original Android platform, apps need to specify all the permissions they need in `AndroidManifest.xml`. During the installation, users will be asked whether they want to grant the requested permissions. Once granted the permissions, an app will have those permissions until it is uninstalled [80]. When the app requests protected resources, the corresponding permission will be checked.

To support component-level access control, we need to attach a separate permission set to different components. Consider all the participants involved in app management, We provide component permission configuration in two different ways. First, an app developer

can specify what permissions each component needs. This is done by adding a special section in `AndroidManifest.xml`²:

```
1 <uses-permission android:name="READ_CONTACTS"/>
2 <uses-permission android:name="INTERNET"/>
3 <package-permission android:name="com.google.ads">
4   <assign-permission android:name="INTERNET"/>
5 </package-permission>
```

In the above permission definition, the app's permissions are defined in Android's original `uses-permission` tag. The app has the `INTERNET` and `READ_CONTACTS` permissions. We call these permissions the app's default permissions. For the components that are not specifically mentioned in `AndroidManifest.xml`, they will have app's default permissions. If app developers want to restrict the permissions of some third-party components, they need to specify that using our new tag called `package-permission`. In the example, we have assigned only the `INTERNET` permission to the `com.google.ads` component. This component can use the Internet, but will not be able to read the user's contact data.

Second, experienced users have options to set their own security policies on a component, putting a restriction on what permissions it can have (see Figure 4.2), regardless of what are assigned to the component during the installation. This is very useful for users to control the popular components, such as advertising components, social networking service APIs, etc. Meanwhile, inexperienced users can keep apps' default settings configured by the developers. Users set component permissions per app. For example, if a user assigns only the `INTERNET` permission to the `AdMob` package in one app,

²For formatting reasons, permission names in our examples are shortened by removing their prefix. For example, the full name of `READ_CONTACTS` should be `android.permission.READ_CONTACTS`.

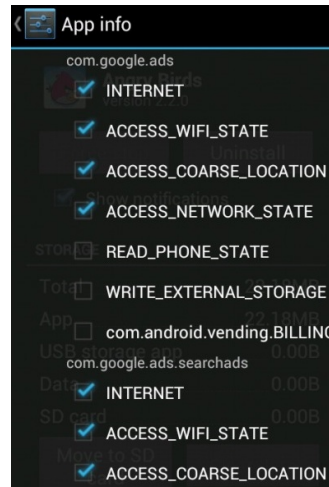


Fig. 4.2.: Permission Manager Interface for Users: third-party components with less permissions will be displayed in system settings, and experienced users have options to adjust component permissions in the app info section of system settings.

then it will not affect the other apps' AdMob component permissions. This setting only overwrites the permissions assigned to the component in this app.

All the app permission definitions will be stored in framework reference monitor as what original Android does. However, all the component permission definitions will be acquired by framework policy manager. Once any component permission update happens, framework policy manager synchronizes the information of the component and its permissions with kernel policy manager. The synchronization is only done when the app is installed/updated, or when users modify the permissions assigned to a component.

4.3.4 Extracting Component Call Chains

To enforce component-level access control, Android needs to know what component initiates the access in addition to the UID information. However, that is not enough, as one component A can invoke another component B, and B then invokes C, which initiates

the access. To make a correct access control decision, Android needs to know the entire call chain $A \rightarrow B \rightarrow C$, instead of C alone. This call chain is called the component call chain in this work. Since Dalvik supports multi-thread, the call chain must be extracted at the thread level, one per thread. When a new thread is spawned, its initial call chain is inherited from the parent thread. The new thread has to keep the initial call chain during its lifetime to avoid escalating its privilege.

The component call chain needs to be extracted at runtime from inside Dalvik. Dalvik functions as the Java interpreter in Android; it converts Java bytecode in dex format into native code and executes the native code [81]. To extract the call chain, we put hooks in Dalvik.

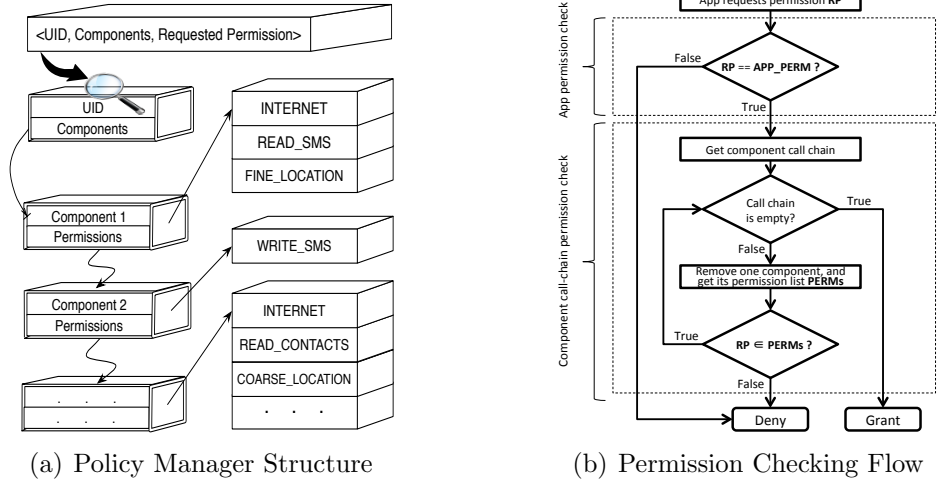


Fig. 4.3.: Policy Manager (PM) Structure & Permission Checking Flow

Hooks

Dalvik’s core interpreter is called `mterp` [82], which interprets the machine-independent bytecode to machine-dependent code. There are many opcodes in bytecode, such as conditional, mathematical, method operations, etc., but component transitions (i.e., from one component to another component) can only happen at the method invocation and return time, so we only focus on the opcodes related to method invocation and return. We have identified all these opcodes ³, and placed a corresponding hook in each of them. These hooks check whether there is a component transition; if so, record the transition to the call chain.

In addition to the hooks placed in `mterp`, we also place hooks in the Java reflection class and `DexClassLoader` to address implicit method invocation and code injection attacks. We will further discuss these two attacks in Section 5.4.

Call-chain storage

Once call chains are collected, the first question is where to store them. With respect to call chain extraction and permission enforcement, three candidate places are safe to store call chains: Dalvik, the kernel and the `system_server` process. From cost efficiency perspective, the kernel is the best place to store call chains. Suppose call chains are stored in Dalvik, every time `system_server` checks permissions, `system_server` needs to request call chain through IPCs. Similarly, if call chains are stored in `system_server`, whenever

³We place hooks in the following opcodes: `invoke-virtual`, `invoke-super`, `invoke-direct`, `invoke-static`, `invoke-interface`, `invoke-virtual/range`, `invoke-super/range`, `invoke-direct/range`, `invoke-static/range`, `invoke-interface-range`, `invoke-virtual-quick`, `invoke-virtual-quick/range`, `invoke-super-quick`, `invoke-super-quick/range`, `return-void`, `return vx`, `return-wide vx`, and `return-object vx`.

Dalvik updates a call chain, it has to talk to `system.server` via IPCs. These two operations are relatively frequent, and so many IPCs will lead to low performance. By storing call chains in the kernel, these two kinds of IPCs turn to system calls. According to our experiments, we find storing call chains in the kernel boosts the speed of call-chain update and permission checks.

Compac stores call chain based on thread. A process can have several threads running simultaneously, thus can have multiple component call chains, one for each thread. When Dalvik detects a component change, it sends the new component information into the kernel, or asks the kernel to remove one from its call chain, depending on whether it is an invocation or return.

Recording a call chain in Dalvik may introduce a high overhead if it is not properly handled. For performance reasons, we optimize the design in three aspects: first, we do not record all component transitions. We only record transitions among the components specified in the `AndroidManifest.xml` and the user's settings, because only these components can cause permission changes. Second, we delay the call-chain synchronization at Java Native Interface (JNI). Dalvik does not synchronize call chain with the kernel every time when there is a component (package) transition. Instead, it does that in JNI. This is because the Java code will eventually be interpreted to native code provided by Android, and JNI is the only entry point where the transition from Java code to native code happens. Third, method invocation within the same component will not be recorded. For example, if a method in package A invokes another method in the same package, which in turns invokes a method in package B, the call chain will be " $A \rightarrow B$ ".

4.3.5 Access Control Enforcement Based on the Call Chain

Compac's permission model is composed of app permission check and component permission check (see Figure 4.3(b)). Compac keeps a clear and standalone design for the two permission checks. Compac remains the app permission check logic in RMs without modification, meanwhile, Compac has two new modules called *policy managers* and *kernel policy manager* to check the component permissions. The two Policy Managers (PMs) are built in the two Reference Monitors (RMs) separately. When a permission request comes to RMs, RMs first check app permissions. If the app does not have the specific permission, the request is denied immediately without going to PMs. Otherwise, RMs ask PMs to begin the component permission checking procedure. PMs send a privileged system call to call-chain storage in order to request component call chain. Once they receive the call chain, they calculate the call-chain permissions and check whether the caller's call chain have the requested permission. PMs return the result to RMs, no matter what the result is. After that, RMs handle the rest as what they do in original Android.

We name the permissions calculated from component call-chain permissions as the *effective permissions*. In the original Android framework, there is no such call chain, so the effective permissions are the same as the app's permissions granted during the installation. In Compac, the effective permissions are calculated as the following definition:

Definition 4.3.1 Effective Permission. *Let C_1, \dots, C_n be components, and P_i be the permission set for the component C_i , where $i = 1, \dots, n$. Assume that the current*

component call chain for a thread is $C_1 \rightarrow C_2 \rightarrow \dots \rightarrow C_n$. The effective permission set eP of the current thread is defined as the following:

$$eP = P_1 \cap P_2 \cap \dots \cap P_n.$$

If $P_{request} \in eP$, PMs allow the access request. In the actual enforcement, PMs do not calculate all the permissions, instead, PMs examine the call chain and just check each component to see if they have the requested permission.

The policy seems too restricted, however, it is effective and practical. First, all the components' permissions of an app are defined by one app developer. Consider the example in Figure 4.4, there are three components C_1, C_2, C_3 , and they have permission sets $\{P_1\}$, $\{P_1, P_2, P_3\}$ and $\{P_2\}$. In case 1, when C_1 calls C_2 in order to use P_2 , the access is denied. If the developer would like C_1 to use P_2 , the developer will assign P_2 to C_1 just like C_3 in case 3. Otherwise, C_1 is trying to gain P_2 without user consent. So, the deny decision is correct. In case 2, we can see the policy allows the components with less privileges to call components with more privileges. As long as the privileged action is not requested, PMs will not check component permissions. Second, the call chain only records components (packages) that are specified with tag `package-permission` in `AndroidManifest.xml`. These are untrusted third-party libraries and there are not too many cross references among them. We conduct several experiments on Compac by recording third party components and logging the call chains. We never see the call chain having more than five components. We also evaluate 34 apps (downloaded from Google Play) by restricting all the third-party components, and we never experience false positives.

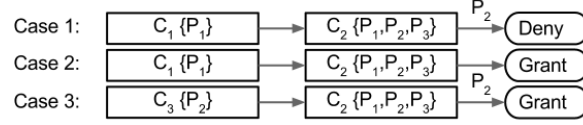


Fig. 4.4.: Component Intersection Cases

4.3.6 Policy Manager Implementation

Policy managers play a critical role in component call-chain access control. Thus, we take FPM as an example to illustrate the internal work flow in PMs. After app permission request is granted, the permission request comes to FPM and FPM begins to check component call-chain permissions. First, FPM gets the caller's identity from Binder IPC, In this case, the caller's identity is thread ID (TID). With the TID, FPM requests and gets the thread's component call chain from call-chain storage. Besides, FPM also has the caller's UID, so FPM can map out the app's components with their permissions from FPM's internal three-level hierarchical data structure, as shown in Figure 4.3(a). KPM has a similar checking procedure except that KPM gets the call chain much easier, because there is no IPC and context switch involved.

During the FPM permission checking procedure, the most difficulty is how the FPM gets the caller's TID since there is no TID related support in original Android. It is much easier if the caller voluntarily sends the TID to FPM. However, such information cannot be trusted. We choose to let the kernel provide the caller's TID. We allow the FPM to call `getCallingTid` API, which is similar to the system's `getCallingUid` API. FPM can use `getCallingUid` to request UID from binder's kernel driver when FPM needs UID to check app permission. In our implementation, we add the `getCallingTid` API support in

Binder’s kernel driver by recording the caller’s TID when IPC happens. We also build both the Java and native `getCallingTid` interfaces for any IPC that goes through binder. As a result, FPM can get the caller’s TID through binder object for authorization.

4.4 Potential Attacks & Defense

The traditional JVM has its own built-in access control [83], which is enforced through the security manager, access controller, and security packages in `java.security.*`. This access control in general falls into two categories: the access control on resources (such as file read/write, hardware access, etc.) and the access control on language properties (such as whether or not a program can use Java reflection or class loader).

To contain native code, Android relies on reference monitors to protect privileged resources and removes access control entirely from Dalvik. Without this kind of access control, it is difficult to achieve the isolation among components. For example, reflection can be used to inject code in another Java package, blurring the boundaries among components. Compac heavily relies on components, so we need to clearly identify component boundaries. In this section, we describe possible attacks on Compac and explain how we remedy these attacks in our design. In a nutshell, we need to compensate for the missing language security feature in Dalvik, in order to secure the component-level access control.

4.4.1 Implicit Invocation Attack

Attack. Reflection is a powerful Java language feature, which allows a piece of code to invoke any method of a class, including its private and protected methods, unless the method is marked as inaccessible for reflection. Based on our study (Table 4.1)⁴, we can see that around 60% of benign apps use reflection to implicitly invoke other methods. Some usages are made by the advertisement code included in apps. After excluding that factor, about 42.49% of benign apps use reflection. The widely used reflection functionality drives the needs to handle this type of special invocations instead of simply blocking them.

Table 4.1: Implicit Invocation Usage (Total App Samples: Benign 16000, Malware 2566)

Implicit Invocation App Sample		Number	Percentage
Benign Apps	including Ads	9577	59.86%
	excluding Ads	6798	42.49%
Malware Apps	including Ads	1377	53.66%
	excluding Ads	989	38.54%

Defense. Compac can easily solve the reflection problem. The Java code is interpreted in Dalvik and Dalvik can trace all the methods including reflection methods. So Dalvik knows which method will be invoked in a reflection invocation. Reflection’s implicit invocation is implemented in the `reflection_native()` method, which resides in the core libraries of Dalvik. We place a hook in `reflection_native` to monitor which Java method will be invoked by reflection, and thus extract the Java package containing the real invoked method instead of reflection packages.

⁴Benign apps are downloaded from Google Play, and most malware apps are collected from Android malware genome project [84].

4.4.2 Inter-Component Code Injection Attacks

If a package can modify another package's code, it can break Compac, because a package with less privileges can simply inject its code into a package with more privileges. We call such an attack the *inter-component code injection attack*. The Java instrumentation class, `java.lang.instrument` [85], can be used to modify the contents of an existing class; fortunately, it has been disabled in Android SDK. Instead, Android develops its own instrumentation package, `android.app.Instrumentation` [86]. This package can be disabled in `AndroidManifest.xml`. It seems that we are safe, but unfortunately, in addition to instrumentation, there are two other ways to modify other class's code.

Attack 1. Reflection can be used to modify the field of a class object (see the following example):

```
1 import java.lang.reflect.Field;
2 Field field = classInstance.getClass().getDeclaredField(fieldName);
3     /* Allow modification on the field */
4 field.setAccessible(true);
5     /* Set the field to a new value */
6 field.set(classInstance, newValue);
```

The field itself can be an object of any type. If the object field is changed, the instance is modified. Moreover, if a class object is changed, the corresponding methods are changed accordingly. Initially, we decide to simply block `set()` method, but our study (Table 4.2) has shown that about 15.24% (12.03% if ads are excluded) of benign apps use the reflection in this way.

Table 4.2: Reflection for Code Injection Usage (Total App Samples: Benign 16000, Malware 2566)

Code Modification App Sample		Number	Percentage
Benign Apps	including Ads	2438	15.24%
	excluding Ads	1925	12.03%
Malware Apps	including Ads	56	2.18%
	excluding Ads	17	0.66%

Attack 2. The second attack is related to the class loader. If developers restrict an untrusted component’s permissions, the untrusted component can escape the restriction using `DexClassLoader`. Using the class loader, an untrusted component can reload a class [55], and therefore can replace a more trustworthy component with its own malicious code. This completely defeats the component-level access control. Class reloading is not possible in JVM, as special permissions need to be granted to an app before it can load classes. Since Android’s Dalvik removes this access control, class reloading becomes possible.

When using class loader `DexClassLoader` in Dalvik, the protected method `loadClass()` in Dalvik does check whether the class is already loaded or not; if it is, it will not reload the same class. However, this is only enforced inside `DexClassLoader`, so if a malicious component extends `DexClassLoader` and overrides `loadClass()`, it can successfully reload a class.

Defense for attacks. For both attacks, we enforce the following policy in Dalvik: if code in package A tries to modify/reload a class in package B , this action is only allowed if $P_B \subseteq P_A$, where P_A and P_B are the permission sets of package A and B , respectively. In

other words, a component with less privileges cannot modify/reload a class (component) with more privileges, and thus cannot gain privilege.

4.4.3 Package Forgery “Attack”

Since Compac identifies components using the Java package name, a potential attack is to forge the package name, so the restriction on the package can be circumvented.

However, this is not a feasible attack. When developers intend to include a third-party package in their apps, they have the responsibility to ensure the integrity of the package.

For example, if they plan to include Google advertisements in their apps, they need to ensure that the AdMob SDK they use is indeed from Google.

4.5 Case Studies & Evaluation

In this section, we conduct two types of evaluations. First, we use three case studies (Ads APIs, social networking service APIs, and web apps) to demonstrate how the privileges of components can be restricted using our component-level access control. In all the case studies, we focus on the most representative permissions, such as `INTERNET`, `READ/WRITE/SEND_SMS`, `READ/WRITE_CONTACTS`, `READ_PHONE_STATE`, `ACCESS_COARSE/FINE_LOCATION`. Second, we evaluate the performance of Compac.

4.5.1 Advertising APIs

We would like to evaluate how Compac works with various advertising packages. Certain advertising APIs (like InMobi) may use user’s phone state or location information

for displaying more relevant Ads to the user. To protect clients' privacy, a developer can prevent the advertising components from using these permissions without harming the app functionality and changing the app's permission set.

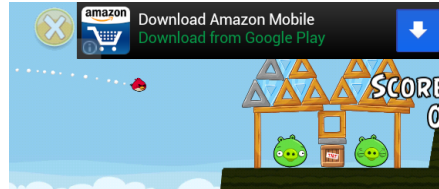


Fig. 4.5.: Angry Birds

We use the Angry Birds app [87] to demonstrate the aforementioned scenario. Angry Birds uses five advertisements including AdMob, InMobi, Millennial Media, JumpTap and GreyStripe. In Angry Birds, advertisements display at the top of the screen (see Figure 4.5), and the app randomly chooses one advertisement from the five to display. We assign only two necessary permissions for the five Ads APIs, while the Angry Birds app has six default permissions. Component permission tags are defined in `AndroidManifest.xml` (to save space, we only show the tags for Google Ads).

```

1 <package-permission android:name="com.google.ads">
2   <assign-permission android:name="INTERNET" />
3   <assign-permission android:name="ACCESS_NETWORK_STATE" />
4 </package-permission>

```

We repackage app's APK file and run the Angry Birds game. As the game runs, we suddenly receive a pop-up window (Figure 4.6), indicating that the developer has not declared the `READ_PHONE_STATE` permission. From the logs (Figure 4.7), we can see that the JumpTap package throws an exception, but the game does not crash and continues running

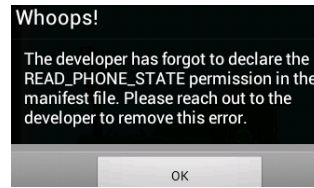


Fig. 4.6.: Angry Birds: No Permission Pop-up Window

```
D/dalvikvm( 663): DALVIKHOOK DEL pkg=com.jumtap.adtag; method=JtAdView.setAdViewListener|
StackSize=0
D/dalvikvm( 663): DALVIKHOOK ADD pkg=com.jumtap.adtag; method=JtAdView.loadUrlIfVisible|S
tackSize=1
D/PackageManager( 81): checkUidPermissionTid android.permission.INTERNET is granted, uid
=10037, tid=663
D/PackageManager( 81): checkUidPermissionTid android.permission.READ_PHONE_STATE is deni
ed, uid=10037, tid=663
E/JtAd ( 663): JtAdManager: Requires READ_PHONE_STATE permission
D/JtAd ( 663): Base url : http://a.jumtap.com/a/ads?textOnly=f&ua=Mozilla%2F5.0+%28Lin
ux%3B+UK3B+Android+4.0.4%3B+en-us%3B+Full+Android+on+Crespo%2FIMM76I%29+AppleWebKit%2F534.3
```

Fig. 4.7.: Ads in Angry Birds: Permission Deny Logcat

smoothly; this is because `READ_PHONE_STATE` is an optional permission and Ads handle it properly. Experienced users can achieve the same goal by modifying the package's permissions in the app setting.

Besides, we download 33 apps from Google Play and run the extreme experiments like giving empty permission set to an app's Ads. The results show that we can successfully restrict all the Ads in these apps. 29 apps handle the no-permission exception, so they continue to run without any problem, except that no advertisement is displayed. 4 apps display "no INTERNET permission" toast messages and then exit.

4.5.2 Social Networking Service APIs

Android apps use a number of social networking service (SNS) APIs, such as Facebook, Twitter, and Dropbox APIs. To use these APIs, apps can include the API packages in the program, and interact with the classes in the packages through the APIs. Once included, the API package will have the same privileges as the app. Unfortunately, some of the

```
java.lang.SecurityException: Permission
Denial: reading com.android.providers.
contacts.ContactsProvider2 uri content://
com.android.contacts/contacts from
pid=637, tid=637, uid=10035 requires
android.permission.READ_CONTACTS
```

Fig. 4.8.: Facebook: Permission Deny Toast

packages seem to abuse the privileges by collecting private information about users. It will be more desirable if apps can limit the privilege of these API packages.

The component-level access control in Compac can be used for this privilege-restriction purpose. To demonstrate that on SNS APIs, we emulate a scenario where a malicious SNS library attempts to read user contacts. We insert a small piece of malicious code in Facebook SDK. As long as the app that uses this API has the permission to read user's contacts, the inserted code can silently read the contacts from the phone and send them out, without user consent. We package the malicious Facebook SDK in a sample app. This app gets the information from the user's friend list (in Facebook), compares the list with the user's contacts (on the device), and sees whether any friend is on the contact list. This is a typical use of Facebook APIs. Without the protection from Compac, the privilege for reading contacts may be abused by the malicious Facebook APIs.

To block the `READ_CONTACTS` permission, we restrict the permissions of the Facebook SDK component to `INTERNET` only. When we run the app under Compac's protection, the app throws a "Permission Denial" toast for `READ_CONTACTS` (see Figure 4.8). As shown in Figure 4.9, the action (reading contacts from phone) has been blocked. The main app still has the `READ_CONTACTS` permission, so we can tell the blocked event is caused by the Facebook component.

```

D/ContentProvider( 231): enforceReadPermission uid=10035, tid=637
D/PackageManager( 79): checkUidPermissionTid android.permission.READ_CONTACTS is denied, uid=10035, tid=637
E/DatabaseUtils( 231): java.lang.SecurityException: Permission Denial: reading com.android.providers.contacts.ContactsProvider2 url content://com.android.contacts/contacts from pid=637, tid=637, uid=10035 requires android.permission.READ_CONTACTS

```

Fig. 4.9.: Facebook: Permission Deny Logcat

4.5.3 Web Applications - PhoneGap

Compac can also protect web apps, although component hooks are not deployed in WebKit's JavaScript interpreter. To demonstrate that, we have conducted experiments on a popular cross-platform web app framework called PhoneGap [88]. PhoneGap encapsulates a WebView, allowing developers use HTML5, JavaScript, and CSS to develop mobile apps. The JavaScript APIs provided by PhoneGap can access device resources through WebView's `addJavascriptInterface` API [35]. However, this API has caused many security problems [89] because it is exposed without control. A recent article [90] demonstrates that JavaScript code can use reflection to gain the Java object reference, thus can gain the app's privilege. The security problem is because WebView cannot contain the JavaScript code.

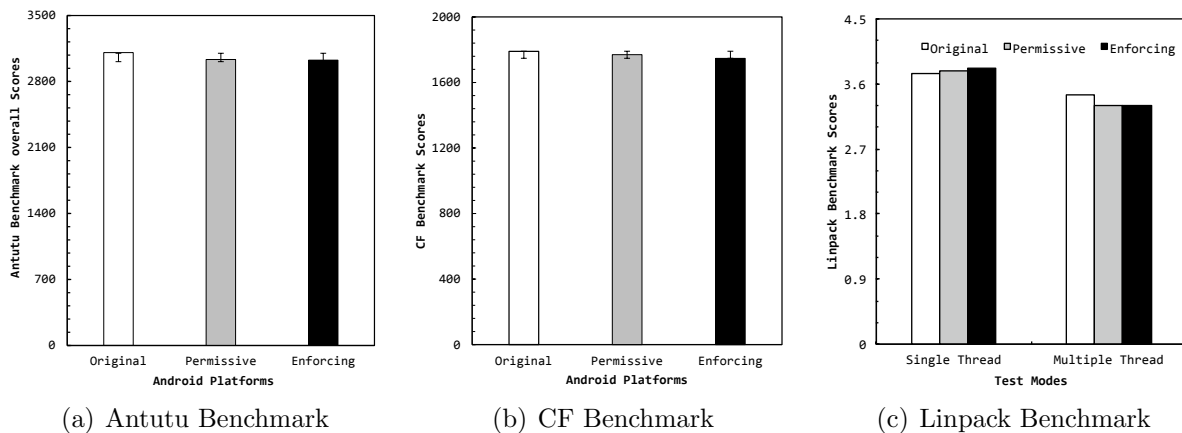


Fig. 4.10.: Overall Performance: Benchmark Results

Compac easily fixes the above over-privilege problem by assigning only the `INTERNET` permission to `WebView` component. Every time the `addJavascriptInterface` API uses reflection to access privileged Java API, it is recorded in the call-chain. When the privileged Java API tries to perform privileged actions on behalf of the JavaScript code, access is denied; If the Java API is called by the app's code and the `addJavascriptInterface` API is not in the call-chain, access is allowed.

Besides, Compac also provides a solution to enforce the principle of least privilege on PhoneGap plug-ins. In PhoneGap, a plug-in usually conducts a particular functionality, such as reading/sending SMS, using camera, etc. All plug-ins are included as libraries. Once included, they have the same privilege as the app, leading to an over-privilege problem. Since all the plug-ins have their own unique packages, third-party plug-ins can be considered as components. Based on the functionalities of plug-ins, we evaluate six different plug-ins including `com.seltzlab.mobile`, `com.leafc-ut.ctrac`, `com.rearden`, `org.devgeeks`, `com.karq.gbackup`, `com.practicaldeveloper.phonegap.plugins`. According to their documents, they require permission `GET_ACCOUNTS`, `CALL_LOG`, `READ_CONTACTS`, `READ_PHONE_STATE`, `READ_SMS`, `SEND_SMS` individually.

We conduct two experiments on these plug-ins. In the first experiment, we remove all the permissions from the plug-ins. We have observed that Compac is able to limit the behaviors of five plug-ins and their requests to access the device resources are all denied, The only exception is the Contact View plug-in (`com.rearden`), which needs no permission and does not perform privilege action directly. Instead, it displays the contacts by sending Intent to Android's built-in Contacts app. This is a privilege escalation problem [40] between apps and out of this work's discussion. In our second experiment, we assign only

the required permission as their documents describe to the six plug-ins. Our results show that all the plug-ins work properly. This indicates that they do not need extra permissions. By combining the two experiments, we have demonstrated that Compac can resolve the over-privilege problem associated with the PhoneGap plug-ins, without affecting their functionalities.

In conclusion, we demonstrate that Compac can successfully put access control on WebView and other web frameworks, and this is not easily done by previous work.

4.5.4 Performance

The overhead of Compac mainly comes from two sources. One is from the hooks that we insert in Dalvik, and the other is from the component permission checks (at both framework and kernel levels). Component permission checks are more complicated, as the effective permissions need to be calculated from the component call chain at runtime. Dalvik hooks just record the necessary Java packages. So, the cost from permission checks is far more than that from Dalvik hooks. We first evaluate the permission check overhead under an extreme condition, which provides a guideline on the upper boundary of Compac's performance. Then we evaluate the overall performance, which illustrates the overhead in normal situations. We employ a Nexus S phone as the evaluation platform. Our codebase is the `Android-4.0.4_r1.2` branch from the Android Open Source Project (AOSP). The original Android and Android with Compac are originally from the same copy of the source code.

Permission Checking Overhead

We first measure the overhead of Compac’s permission checks. In Android, As we know, permissions are checked either in the kernel through system calls, or at the framework level through IPC (IPC permission checks). The cost of an IPC permission check is always higher than a kernel permission check, because an IPC involves several system calls. Therefore, we only perform evaluation on IPC permission checks to measure the upper boundary of overhead caused by Compac’s access control. We use the following pseudo-code to show how we conduct the unit test.

```

1  time_start := System.currentTimeMillis
2  for i:=1 to 25000 do
3      check random permission
4  time_end := System.currentTimeMillis
5  excute_time := time_end - time_start

```

We do the test by conducting the IPC permission checks on random permissions 25000 times. On the original Android, the test takes 38,153 ms, and on Android with Compac, it takes 46,614 ms. Therefore, Compac’s overhead for IPC permission checks is 22.2%. Although this number seems high, it just provides the upper boundary of the Compac’s performance and does not pose problems for the overall performance. Because the IPC permission check happens only when a privileged action is triggered. A normal app does not have many such IPC permission checks.

Overall Performance

To evaluate Compac’s overall performance, we use three popular benchmark apps: Antutu, CF-bench, and Linpack. We plot the results in Figure 4.10. In the figure, “Original” means the AOSP Android; “Permissive” means Android with Compac will be in permissive mode, i.e. it will log access control denials but not enforce them; “Enforcing” means we restrict each package’s permissions inside the benchmark apps and Android with Compac runs in enforcing mode.

The three benchmark apps show similar results that the overall performance of Compac is quite close to the original Android platform. A higher number is better in all the three benchmarks. Take the Antutu benchmark app as an example. Antutu produces an overall score based on the measures. The result shows that the original Android score is 3106, Compac in permissive mode scores 3033, and Compac in enforcing mode scores 3026. Taking the original Android score as the base, the performance of Compac in permissive mode is 97.6% and the performance of Compac in enforcing mode is 97.4%. One interesting thing is that for the single thread test with Linpack, Android with Compac has a higher score than original Android. We run the experiment several times and get similar results. Sometimes, Compac in permissive mode may score better than Compac in enforcing mode, but both always score better than original Android. The reason is we optimize Dalvik interpreter and related libraries when we implement Compac. According to Linpack for Android app’s description [91], “the Dalvik VM has a huge impact on the Linpack number”, and “this test is more a reflection of the state of the Android Dalvik Virtual Machine than of the floating point performance of the underlying processor.

Software written for an Android device is written using Java code that the Dalvik VM interprets at run time”.

4.6 Related Work

Information Flow Tracking. To protect user privacy, the pioneering work TaintDroid [67] conducts taint analysis in Dalvik to trace information flows. It has the ability to trace when the sensitive information flows out of an app. TaintDroid taints data in Dalvik while our work builds hooks in Dalvik for system to make access control.

AppFence [68] is built upon TaintDroid and can block unwanted data transmission.

Android Permission Control. A collection of work [29–32, 69] adopt different policies to achieve fine-grained access control on app permissions in order to control an app’s behaviors or make sure the app has the least privilege. Apex [29] and Kirin [32] allow users to accept a subset of the permissions declared by apps and enforce policies at installation-time. Saint [31] enforces policies at both installation-time and runtime, and the policies leverage the relationship between the caller app and the callee app. The user-driven work [69] considers certain user inputs as contexts. Upon occurrence of certain user actions in special UI components, the system grants corresponding permissions to the app. CRePE [30] uses environments as contexts to control the behavior of an app.

In-App Reference Monitor. Aurasium [66] and previous work [11, 13, 14] build a reference monitor within an app to achieve access control through code instrumentation or rewriting. They can flexibly enforce app level policies but may not enforce component-level policies. Because all the code as well as the reference monitor run in the same process.

Without jumping out of Dalvik or the process, it is difficult for the reference monitor to recognize the running code. Besides, the current Android does not provide component information for access control either. However, Aurasium can still be used by app markets to protect users from malicious and untrusted apps if app markets wrap each app with a reference monitor and consider the whole app as the target.

Mandatory Access Control. Previous arts XManDroid [24], TrustDroid [36], IPC inspection [40] and Quire [34] deal with the privilege escalation problems across different apps, while Compac focuses on the components inside the same app (same process). They are the first work to propose mandatory access control concept in Android. SEDalvik [92] also intercepts the Java methods in Dalvik as Compac, but it enforces policy within Dalvik. SEDalvik can prevent some Java level malware, however, the system cannot be aware of the security contexts, and Dalvik cannot understand the contextual information of the code. So, SEDalvik turns to a language-level mandatory access control within apps.

SEAndroid [23] implements SELinux in Android kernel and builds middleware mandatory access control in both the kernel and Android framework. SEAndroid ports the core SELinux into Android’s kernel and makes all the processes, files, sockets and other kernel resources under the control of the MAC. Thus, SEAndroid limits root and other users’ privilege. SEAndroid’s install-time MAC, intent MAC and content provider MAC provide a comprehensive protection to apps and Android middleware. SEAndroid and our work strictly follow the same design principle (least privilege principle). SEAndroid divides the privileges of the system, users and resources, while our work divides the privileges of the apps.

FlaskDroid [26, 65] extends the type enforcement policy language from SELinux to Android middleware and offers API-oriented MAC control for multiple stakeholders including app developers. FlaskDroid allows developers and users to have finer-grained access controls in services and content providers. For example, the content in content provider can be partially displayed according to the policy. FlaskDroid makes Android middleware be aware of the contextual information in services and content providers, and our work makes reference monitor get the component contexts of an app.

Component-level Access Control. AdDroid [1] isolates advertising components by encapsulating them as a service and creating ADVERTISING permission for the advertising service. AdSplit [2] enforces a fine-grained access control by putting the advertising component entirely in a separate process, which relies on the process-level isolation to achieve the protection. This solution implements a strong component isolation mainly for components like advertisements, which need less or no communication with other components. AFrame [3] provides a general approach to isolate third-party libraries into a separate process. It prefers libraries that need strong isolation and do not interact quite often with the main process while Compac is used for those libraries work tightly with other components.

4.7 Conclusion

To reduce the risks caused by the untrusted third-party code included in Android apps, we propose Compac, a generic approach to achieve component-level access control. Compac extends the existing Android security model. It uses Java package as component and enforces access control based on the UID and component. Using Compac, a component in an app can be given a subset of the app's permissions. We conduct case studies on various third-party APIs including advertising, social networking service, PhoneGap plug-ins, and WebView to demonstrate its effectiveness, usefulness and the compatibility with the existing Android architecture. We also evaluate its performance to show that the overhead is quite affordable. The source code is available at <https://bitbucket.org/syr/compac>.

5. Understanding and Mitigating Security Risks in Host Card Emulation (HCE)

5.1 Introduction

Mobile wallets have become a more and more important payment channel. For example, Android pay has already gained attentions recently, and starts to serve millions of users worldwide. By statistics, Android payment has been launched in more than five countries, support 100+ bank cards, and accepts 100,000+ locations worldwide [93]. A study [94] predicts that the transaction value of android payment in the U.S. could increase to more than 140 billion U.S. dollars by 2019.

All the wallets are using different form factors like NFC, QR code, MST and so on. Among which, NFC is widely used for processing card payment. Mobile wallets like Apple Pay, Android Pay, Samsung Pay, Microsoft Pay and numerous of banks' digital wallets like Capital One Wallet [95], Wells Fargo Wallet [96], have been developed and adopted widely. Specifically, most of the NFC-based mobile wallets are built on Android platform, the most popular mobile OS.

As mobile wallets introduce EMV payment method, it looks like those wallets are secure enough from attacks like man-in-the-middle attacks or Mafia attacks. However, our study shows that may not be the case on Android, due to the mechanism of HCE service. EMV-based protocols employ nonce on the communication between Point Of Sales (POS)

and NFC payment cards, making it difficult to attack real-time transactions. In addition, radio jamming and hijacking a transaction session makes the attack cost extremely high, as a result, it is not common in the large wild. Our study of HCE service find that, credit cards in a HCE-based wallet like Google Pay can be tricked by attackers for unauthorized payment, and the attackers only needs system privilege by leveraging existing Linux Kernel CVEs.

Prior to know more about our findings, an overview of existing HCE architecture needs to be explained. Android's solution of NFC payment is based on their own HCE architecture. Such architecture separates the hardware control and data logic. A dedicated NFC service controls the hardware (a.k.a NFC controller), and all the data in/out are routed to NFC service. To enable 3rd party wallet, Android defines standard HCE service template and any wallet extends such service template will be recognized and displayed to user for selection. In other words, 3rd party wallet's HCE service is called as callback by the NFC service when needed to process transaction data. Such binding callback service model enables 3rd party apps to be service providers while keeps hardware or system resources under the control of the system. Other major services, like accessibility, carrier, device admin, input method, voice interaction, VPN, printer, etc, are implemented in the same way.

As core service providers, such 3rd party services should not be accessed and controlled by others than the system. Thus, Android predefines permissions and forces such services to declare the corresponding permission as the prerequisites to be use on a device. In this way, only the system can access such 3rd party service providers.

A big problem of such model is to treat all 3rd party providers with same access control level. Android permissions provides the same protection level for such service providers and other system resources. So, serving as device admin, VPN, and printer, might be good with permission model's protection. However, payment service like HCE service usually requires a higher standard of protection. For example, financial data in a chip credit card are processed by a tamper-resistant secure element chip. Once the circuit is modified, data cannot be retrieved. Once a the system is comprised, Android is not capable of protecting such financial data. Our study demonstrates that one can steal credit card data for payment with a zero day attack. To raise the security bar for such 3rd party services, protection beyond system level is required. By leveraging EL3 technologies, we develop a solution to protect 3rd party service providers to meet their industrial standard. Our work can allow only a specified component to access such providers. We make a proof-of-concept to protect HCE service, and it can extend to other similar binded service providers as well.

Attacks. We study Host Card Emulation technology and similar binded service models on Android. Our study reveals that HCE is more vulnerable compared to previous payment technology due to an overlooked and underestimated binded service model's weakness, affecting millions of creditdebit cards enrolled and provisioned on Android with HCE-based digital wallets. Such attack hijacks a card's payment session by directly binding to the HCE service hosting such digital card, thus, we call it **Host Card Session Hijacking** (HCSH). The HCSH attack can be issued remotely with high invisibility, causing significant economic loss of the user. Instead of directly stealing a user's payment cards or credentials, the idea of HCSH is to hijack a payment transaction session and make payment remotely without user's consent, so as to spend the victim's money indirectly. No

matter if a card is provisioned as a token or not, both EMV and Non-EMV transactions are affected as long as the digital wallets use HCE technology on Android.

HCSH can be asynchronized or synchronized depending on the NFC transaction mode. The asynchronized HCSH can target on Non-EMV mode, providing flexible time window of using victim's money. It hijacks the session and redirect the response to a software mimicked point of sale and transmits the entire transaction sequences to remote server. Later, a fake mobile wallets can transmit the transaction with proper adjustment of some fields and successfully make payment.

Synchronized HCSH target on both Non-EMV and EMV modes, providing the broadly coverage of card networks and card types. Even the offline transaction with ODA can be affected. Synchronized HCSH hijacks the session at real time while the attacker, simultaneously, starts a transaction at the remote end. The attacker is the real user standing in front of the merchant and the victim's wallet invisibly run at background without the user's consent during the process of attacker paying. If the mobile wallet supports notification, later, it may receive the notification from card network, but it will be too late by then. The attacker needs to pay only when the remote victim's HCE wallet is in satisfied condition. We developed a scalable method to dramatically increase the satisfied condition window and demonstrate that the attack can be practical in realistic world.

Mitigation. tokens' security is heavily dependent on the secure level of the token container and transmission channel. Without reinforcement, the transmission channel purely relies on the host OS.

As the trend of payment, HCE wallets and tokenization have received lots of praises. Both the security and usability are vital to the future of such new technology. The reason

of producing HCSH attacks is the weakness of binded service model. The binded service like HCE service fails to match the finical requirements with a system restricted only permission, meanwhile, the frequently used SE-based chip card can protect assets against both logical and physical attacks even the high level OS is compromised. To provide a higher security of accessing aforementioned binded services, we propose an approach to strengthen HCE data channel with minimum change to user habit and system compatibility. We realize a more secure layer needs to be introduced as the root of trust for the binded service. To prove our design, we employ TEE technology and implement a proof-of-concept (POC), taking HCE service as a binded service example.

In the POC, only the legitimate system component may have the access of binded services with TrustZone as the root of trust. In HCE service case, the NFC service, together with the only hardware NFC controller, is considered as the legitimate subject to access the binded HCE service providers. The POC implement an encrypted channel between targeted HCE wallet and the NFC controller. and only data recognized by the NFC service can be sent out through the NFC controller. The HCE wallet data and its routing channels are secured by the TrustZone OS, which is standalone with the Android host.

To summarize, we studied Host card emulation technology and pointed out the weakness and security risks. To illustrate the importance of such risks, we demonstrate a series of real-world attacks to remotely manipulate Android Pay wallet, steal and uses payment credentials without being noticed by users. Moreover, we proposed a solution of HCE using TEE technology and implemented a proof of concept on Samsung Galaxy S6. By doing so, we hope to help HCE retain the flexibility and convenience meanwhile

decoupling the security of the Android OS and HCE wallet data to achieve 3rd party service’s protection.

Contributions. The contributions of the work are organized as follows:

- We study HCE technology and its security architecture, and uncover a vital design shortcoming of the 3rd party service providers on Android. To arise the alarm of such shortcoming, we are the first to report a group of HCE hijacking attacks. The attacks are highly invisible and scalable and can be weaponized to cause severe financial loss.
- To make up for the shortcoming, we propose a defense mechanism that can mitigate the threats from within the system. Our analysis showed that system security may not meet the security need of financial transactions, and a more secure environment is needed. In our mitigation, we, taking HCE service as an example, extend TEE protection to 3rd party service providers, and develop a root of trust within the system to improve the access control on 3rd party service providers.

Roadmap. The reminder of this work is organized as follows. In Section 5.2, we provide the background related to our study. In Section 5.4, we detail the steps of achieving the attacks and three attack scenarios. In Section 5.5, we analyze the problem, present our mitigation and evaluates the performance of our defense work. Finally, we discuss the related work in Section 5.6 and conclude in Section 4.7.

5.2 Background

Contactless Payment. Almost all the mobile payment utilizes contactless communication technology due to the lack of contact interface. The most popular

contactless interfaces are Near Field Communication (NFC), QR code, and Magnetic Secure Transmission (MST), etc. In general, NFC is a family member of RFID technology but supports peer-to-peer communication. NFC consists of a collection of industrial or international protocols at multiple layers in the protocol stack. Based on ISO/IEC 14443, NFC in mobile payment operates within an range of up to 10 centimeters at 13.56 MHz. At application layer, smart cards including (mobile payment application) communicates in form of Application Protocol Data Unit (APDU) defined in ISO/IEC7816-4. an APDU adopts tag-length-value (TLV) format, serving as as the basic data unit for the bidirectional communication channel.

EMV and Mag-Stripe Modes. A NFC-based payment usually runs the following two payment modes: EMV and mag-stripe) mode. In EMV mode contactless card builds a more secure mechanism to ensure data security. Running with EMV mode, the card requires unpredictable number (UN), in addition to regular transaction information (amount, currency, etc) from a POS, and the card generates a HMAC-based application cryptogram with the transaction information. In a digital wallet (like Apple Pay, Google Pay), the symmetric key used to generate application cryptogram is protected by all means of security infrastructure and may be frequently refreshed.

In contrast, mag-stripe mode is designed to be backward-compatible in countries like US. Limited by the old mag-stripe protocols used by many existing POSes, each card networks have to capsule tokenized card information such tokens, cryptograms, virtual expiration date into the old fashioned mag-stripe card format.

Host Card Emulation. Initially, mobile wallets employs a small computing chip, like Secure Element (SE), to store and compute data. The chip is wired to NFC contactless

front end (CLF) directly. Such design is to mimic a chip-based contactless card. However, the invasion of SE-based solution for mobile is not good as expected due to the nature of SE. First, access to the embedded SE is limited to few stakeholders. The SE chip is either embedded or inside SIM card. In both cases, only device vendors or mobile network operators have full access to the SE chips. Unfortunately, they do not delegate the privilege of accessing the SE chips to third parties due to security concerns. Second, the development and maintenance lifecycle of SE is relatively long. limited by hardware, the SE applet development has very limited API support for developers, which also limits the spread of NFC utilization to third parties. Android, like many other mobile platforms, introduces Host-based Card Emulation (HCE) technology since Android version 4.4 (Kitkat). HCE is a software-based card emulation architecture providing emulated representation of electronic identity (access, transit and banking) cards by software applications. In HCE, an app, by leveraging system API, can read/write data from/to NFC CLF, turning the app to be an emulated card.

From the architecture perspective, a dedicated NFC process is responsible for delegating the NFC operations to other rich OS applications, routing the NFC data and accessing NFC controller (NFCC) driver in kernel. Specifically, The process is composed of two major components: a NFC system service and the native NFC Controller Interface (NCI) library. As part of Android framework, the NFC system service provides Android App NFC access entry-point and NFC chip management. Meanwhile, the service supports three modes of NFC operations: P2P, read/write and HCE. Acting as NFC HAL interface, the NCI library translates transmission layer protocol and operates the payload in the desired way. In the host OS kernel, the NFCC driver drives NFCC through common

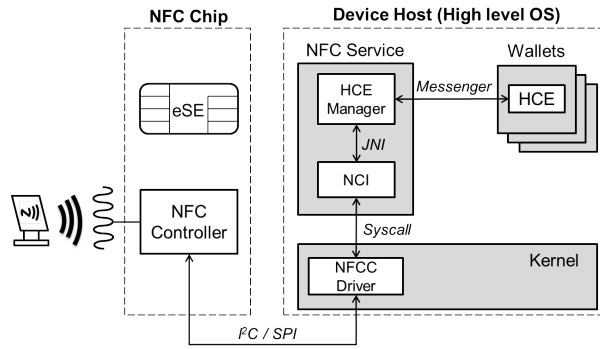


Fig. 5.1.: HCE Architecture Overview

hardware interface such as UART, SPI, and i²O, etc. From the view of security, the NFC data routing is protected by the layer-based access control model. At application level, an app needs to define a HCE service so called **HostApuService** and declare a new system level Android permission **BIND_NFC_SERVICE** in order to take advantage of HCE functionalities. The app can act as a digital wallet and access the NFC system service's NFC data only after the user explicitly grants the permission and sets the app's HCE service to be the default HCE service. Inside the NFC system service, its process is granted with a dedicated kernel GID called `nfc`. which is also the UID of the NFCC driver. Under the protection of host OS kernel access control, normal app cannot bypass NFC system service and access NFCC arbitrarily.

3rd Party Service Provider Model. To step back and have an overview of the HCE service, it actually implements the core digital wallet service as a service provider to the system in order to enable the digital wallet feature. In such model, the core service is provided by the 3rd party and this 3rd party service serve as callback as needed by the system on be half of the user. In some cases, the feature requires some kind of hardware to finish the 3rd party service's functionality. The hardware is binded to the system service,

and the system service decides how to select one 3rd party service from the list. Also, the system service dispatch the request to the 3rd party service. In other cases, the hardware may not be required and the system service solely calls the 3rd party service providers without communicating to the hardware. 3rd party VPN service apps fall into this category.

5.3 OVERVIEW

HCE service as the representative of 3rd party service provider model will be our focus to illustrate the 3rd party service provider concern and propose the mitigation towards this concern. On one hand, the security design of HCE seems impeccable. It is dependent on the permission system, a userspace level access control, and the kernel access control like Linux Kernel DAC or even MAC solution like SELinux. On the other hand, HCE technology, for the first time, exposes a emulated contactless card in a software-based environment. Unlike SE-based solution, a HCE-based mobile wallet runs in the context of rich host OS system shared with all applications. We cannot assume the benignness of apps installed in the same system, nor count on that the mobile user can detect the rooting behaviors and never root their system for advanced features. Thus, the host OS's runtime is much more complicated than ever. The system designer overestimates the complicated host OS context environment and expect to use a system-level permission to handle all special security need of HCE service. **Standing on the point of the system, it is arguable to assume the system is trusted and all the system services and apps run with the above assumption.** Without such assumption, we cannot build security

inside the system. **However, Standing on the point of the contactless card, it really desires a higher security standard like SE solutions with an isolated channel or even tamper-resist capability.** Due to the fact that contactless card is built inside the system, the emulated contactless card's security should be reviewed from the above two aspects. The biggest security concern of HCE , we believe, is the security design principal. It builds a solid security solution from the system's perspective, but underestimates the unique security requirement of emulated contactless card compared to other system services. contactless card requires a sort of isolation with other system context to reduce the threats from the system.

Based on the weakness of the HCE, we develop a new type of attacks targeting on the missing TCB for contactless card. This type of attacks has two variants for EMV/non-EMV digital payment cards or tokens, but both variants have one common feature. With unauthorized access, a malicious app may hijack a transaction session of an emulated card to use the transaction for other purposes, so we name them Host Card Session Hijacking (HCSH) attacks. The attack's footprint is relative small so that the victim cannot easily notice it prior fund loss. It requires no physical sniffer tools, and can be issued remotely and conceals the attack behaviors, thus such attacks reduce the cost of financial cyber attacks. To mitigate such threats, we propose our solutions and implement a prototype based on TrustZone technology.

5.4 The HCSH attack

5.4.1 Attacking Analysis

Our attack is to target the HCE technology introduced in Android. Similar attack may happen on other 3rd party service providers like VPN apps, but the exploit details vary depending on the characteristics of the 3rd party app developers. In the attack, we employ a zero day attack and it seems to be a straightforward thing with system or root privilege considering the large number of CVEs at wild. In this attack our target is not a typical system level service but a 3rd party HCE-based wallet protected by a system permission. Even with system privilege, it is not trivial to attack HCE-based wallets in the following three reasons:

- Payment networks, as well as card issuing banks, employ tokens to process transactions. Samsung Pay stores token information with the help of TrustZone. Google Pay stores tokens and keys separately on device and in their cloud, and utilizes sessions to manage a dynamically generated key in local device.
- Payment protocols including EMV and mag-stripe modes can provide additional security enhancement at application layer. In each EMV transaction, digit wallet generates application cryptogram based on POS and digital card information. Thus, preplayed transactions cannot be retransmitted. In mag-stripe transactions, another kind of cryptogram is still used even the tokenization back-end service.
- Commercialized digital wallets build their own authentication and other possible threats detection mechanism. All the above three wallets have strong fingerprint or

password authentications to protecting transactions. Samsung Pay even use Trusted User Interface (TUI), a technology backed with TrustZone, for password authentication. Transaction requests without being authenticated are denied by the digital wallets.

The proposed HCSH attack attacks the exposed software-based surface of the HCE service, instead of the implementation flaws of any digital wallets. Therefore, all the HCE-based wallets including GPay and Samsung Pay are potential impacted. Apple Pay, employing SE, is out of the scope of the HCSH. HCE technology has or ever been included in Android OS, blackberry OS and Microsoft Mobile OS. In this work, we only demonstrates attacks on Android OS (due to the dominant marking shares) but other mobile OSes' HCE has similar issues. Our goal is not to argue the security of EMV or tokenization standard, but we would like to demonstrate that a weak access control on the service may break the existing security standard, with the help of some side channel tricks. As a common sense, we assume merchants' POSes are benign and will not collaboratively work with the attacker. Otherwise, the problem becomes to an fraudulent POS stealing user's money, and it is out of the scope of the issues discussed here.

In this paragraph, we interpret the main steps of issuing the HCSH attack. As mentioned above, such attack is non-trivial, and we face the above three challenges. The only exploitable thing is the HCE service interface with the standard API defined in Android development documents. Of the three exposed APIs, only the `processCommandApdu` can request and respond data.

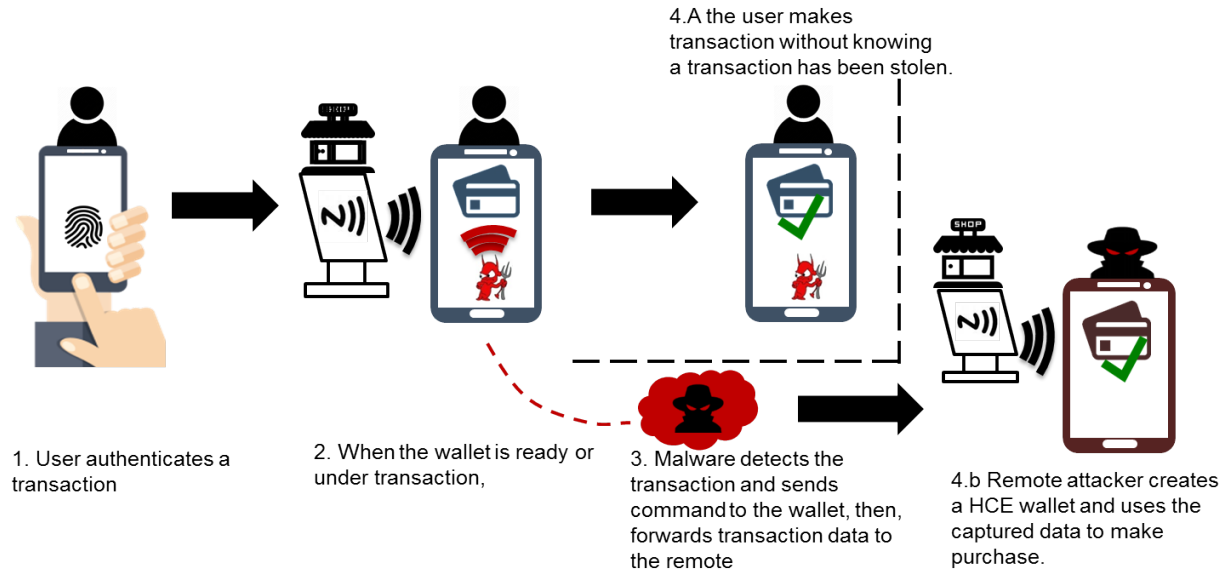


Fig. 5.2.: The HCSH attack flow

We notice that all the contactless transactions are session based. As long as the malware can bypass the payment credentials and take advantage of a payment session, it may manipulate the service for evil purposes. With such guideline, we investigate the user behavior and design the attack flow, as shown in figure 5.2, with five steps. First, the user goes shopping and authenticate her mobile wallet for payment. Second, Disguised as POS, the malware detects the payment window or flood payment request command to the wallet. Based on the response from wallet, the malware adjust request command and keep talking to the wallet until the sensitive transaction information is collected. Third, the malware sends the transaction information to the remote attacker in real-time or later. After that, the user will notice the tap to the merchant's POS succeeds, sometimes, the transaction may fail and the user may retry. The remote attacker will use the collected transaction to craft new transaction based on the new value or fields requirement from attacker side's merchant POS and send the transaction with her own HCE wallet.

The step 3 (Figure 5.2) plays the key role in stealing the transaction. Technically, stealing a transaction is composed of three pieces: Binding to a HCE service, bypassing the wallet's authentication, forging command APDUs and replicating a transaction session. In the rest of this section, we discuss the HCE variants of non-EMV and EMV respectively. The biggest difference between the two variants is the request command sequences (or command state machines) have to be uniquely crafted, targeting a specific attack scenario and card network. Therefore, in the rest of this section, we take non-EMV transaction as an example to demonstrate the technical details of HCSH attacks. After that, we discuss the attacks on EMV only tokens with respect to the various challenges.

5.4.2 Binding to a HCE Service

To steal a transaction pragmatically, the very first step is to connect to the HCE service.

The HCSH attack targets on the tokens enrolled in a HCE-based wallet. As one of the most representative HCE-based wallet app, Google Pay (formed called Android Pay) supports four major card networks (Visa, MasterCard, American Express and Discover) and numerous of banks. In this section, we take Google Pay as an example to demonstrate the significant security impact caused by HCE design shortcomings.

For each payment transaction, there is a session. The session is not only tied to application layer, but also associated with the physical layer for contactless transaction. The physical layer session secures the channel and can prevent man-in-the-middle attacks. If another contactless card or party is in field, the physical layer connection will be interrupted, thereafter, the session will be ended because of the link loss. In HCE, the

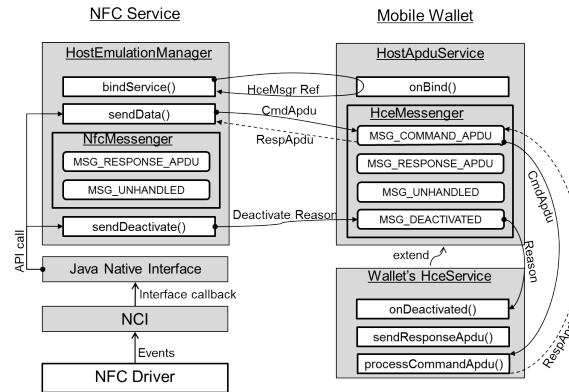


Fig. 5.3.: Bind Service

physical layer entity and application layer entity are separated. The NFC service handles link layer connection, and the digital wallet is responsible for the Android layer. When a link is lost, the NFC service will send a link loss callback to notify the active HCE wallet app.

The separation of the two layers imposes a security risk. Since the application layer connection of HCE wallet is at API-level, HCE service does not differentiate the caller entity. Therefore, HCE service blindly process the incoming command based on its own payment logic. As a result, a third party can inject proper API call to HCE service during an existing payment session, and the digital wallet will process the request and respond sensitive transaction data back to the third party. As a result, neither NFC service or wallet app can detect and prevent such injected commands.

To acquire this permission, the platform signature is required. The fact is Android collects app signatures during the installation but checks the signature by using permanent signature cache file, which is created in `/data/packages.xml` by the system. With the above fact, the malicious app is able to replace its own specific signature with the system

signature tag. The replaced info is loaded to the system after reboots without verifying the app APK's signature and the system will recognize the malicious with system privilege, thereafter, acquire the permission.

With such investigation, we find the attack surface and be able to send data to the apps.

5.4.3 Bypassing individual wallet's Authentication

Although in theory, the above processes should work for the malware, but, in reality, almost all the existing wallet apps implement their own authentication methods to protect the cards enrolled in the wallet. The malware needs to figure out a way to bypass additional security layer in order to initiate a session. The wallet apps may use device-level (such as PIN, device password, biometrics) authentication or network based password to authenticate the users, which increases the difficulty of brute-forcing the authentication methods.

The malware can just wait for the user authenticated the app for legitimate use and catch that session for the evil purpose. The HCE service, protected by the permission, cannot tell who is the caller. As long as the caller has the `BIND_NFC_SERVICE` permission, it allows the access without further discrimination. A common way to tell the authentication status of an app is to take advantage of side channel. Thanks to the HCE service running in the host OS, again, the malware can easily find a side channel to tell the status. Here are the side channels of Google Pay and Samsung Pay, that can be leveraged. Google Pay borrows the system keyguard for authentication purpose. Once authenticated, an authenticated session will last some minutes, although the malware only need a regular

session time which usually around one or two seconds. We monitor screen lock and unlock by checking a public non-privileged API `inKeyguardRestrictedInputMode` in keyguard manager, and we identify that for Android wallet's authentication, the foreground activity switch from a current activity to keyguard activity and immediately jumped to the payment result UI activity of Google Pay. For Samsung Pay, it employs a Samsung only TUI, which is an obvious identifier. For each authentication, this TUI's activity will be foreground and then the foreground process will be switched to Samsung Pay app. Other apps may have similar situation since both the HCE service and the malware are in the same context of Host Android OS. By doing so, we can successfully catch the exact moment of the authenticated window, and deliver the data requests to the wallet apps.

5.4.4 Crafting Command APDUs

In the context of contactless payment cards or tokens, APDU is the communication unit at application layer defined in ISO/IEC 7816-4. Below the APDU application layer, the communication uses electromagnetic induction in order to transmit information. The phone performs as a passive device and its NFC contactless front-end (NFC CLF) receives radio waves from NFC POS and setup the contactless connection based on the physical characteristics defined in ISO 14443 A-1/ISO 18092. The NFC CLF as part of NFC controller set ups a communication session with NFC POS eventually. Since HCE targets on the APDU, the underlying protocol is handled by the firmware and transparent to the HCE wallets and the attacker only needs to focus on the APDU layer.

A NFC payment session is composed of several pairs of command APDU and response APDU. A command APDU comes from the POS and is received by the HCE service. After the HCE service processes the command APDU, it generates its APDU and sends back as the response APDU. HCE service defines four standard APIs but `processCommandApdu` is the only API to receive command APDUs. If the HCE service cannot respond timely, it will reply an empty object indicating that the response APDU will be sent back through another API `sendData` shortly to ask the POS wait for it.

The malware needs to simulate a software-based POS to initiate a session, which requires the understanding of the targeting card networks. As we all know, card network contactless specifications and EMV specs are compatible. Although they might not exactly be the same, but that is really far beyond enough to forge a POS. The POS has different “kernels”¹ to different card networks 5.4.4.

heightKernels	Supported Card Networks
Kernel 1	some JCB and Visa cards
Kernel 2	MasterCard
Kernel 3	Visa
Kernel 4	American Express
Kernel 5	JCB
Kernel 6	Discover
Kernel 7	UnionPay

As defined in EMV contactless Book B, entry points consist of five main functional sections: pre-processing, protocol activation, combination selection, kernel activation and

¹Within these specifications, a kernel is software in the POS System that processes certain contactless transactions

outcome processing. The first two sections get the POS prepared and activate the contactless interface before the transaction starts. In combination selection, the POS sends a static select command by selecting Proximity Payment System Environment (PPSE) and the wallet responds with one or more AIDs that it supports as candidate cards for the transaction; Based on the card network, the protocol and processing procedure may be differently for each card brand. In kernel activation, the POS select one AID and activate the corresponding kernel for the following transactions. In outcome processing, the card and transaction related payment data are exchanged.

The APDU exchange process normally have the following procedure: selectPPSE, select AID, get processing options, read data, generate AC. Different card network may have its own APDU exchange flow, however, in select PPSE command, the terminal always send a static command to initiate the card selection procedure, then the app response with supported AIDs. After that the terminal selects the preferred AID from the list and the card will response with data specific to the AID. So far, the procedure should be common to almost all the payment card networks. Each AID represents to a specific card network, and the POS will active the specific kernel after receiving the response of selectAID. After that, both the POS and the app will talk the specified payment protocols and the APDU commands and responses may vary based on the kernel specifications.

5.4.5 Replicating a Transaction Session

To buy goods for the evil, the malware must be able to replicate the transaction session, which is the last step of the HCSH attacks. To replicate a transaction, we first

need to tell the mode of a transaction: Mag-Stripe or EMV, since the two lead to different transaction replication processes.

Mag-stripe and EMV modes

Mag-stripe mode can be replayed while EMV cannot. Mag-stripe mode is also called Non-EMV mode is compatible by EMV specs, so it is enabled mostly by general POSes and digital wallets, for example, the token in Google Pay support mag-stripe, as well as EMV, modes. EMV mode is more secure than mag-Stripe. For an EMV transaction, a session is tied to the given token and POS. The POS generates a random nonce and the cryptogram from the wallet is associated with nonce, timestamp, amount, etc., in addition to token key and other general card info including but not limited to digital primary number (DPAN), expiry date, or CVV. Some card networks' mag-stripe mode does not require a nonce from the POS, therefore, their generated APDUs in a session can be replayed later.

Due to the feature listed above, the replication time phases are distinct. For the mag-stripe transaction, the malware can cache the mag-stripe transactions, send to other devices and reply later for any other transactions; For the EMV transaction, the malware has to issue a payment remotely and simultaneously with a EMV transaction, but it is still well worth to take. First, EMV transactions usually has a higher success rate by card networks especially for transactions with large purchase amount. Second, a victim's multiple wallets can be attacked by the same time because the each wallet's HCE service is considered as callback of the NFC service without binding number limitation. Third, the

victim group can be a large number, at the moment of the attacker would like to do evil transaction, the chance of finding one target can be high.

By far, the HCSH attacks are successfully issued. During the progress, the victim has no awareness of the attacking behavior, unless some wallets may send the victim notifications later. However, that would be too late for the victim to recover her asserts.

5.5 Mitigation

Our attacks uncovered a significant security gap between the existing contactless requirement and HCE security. There is no argument on the need of security enhancement to HCE architecture due to the important roles that it plays in mobile payment world. Based on the above fact, either a new solution meeting with the contactless requirement is invented or a mitigation of the existing HCE security gap is proposed. As HCE architecture creates a booming mobile payment eco-system, we follow the goal of mitigating its security concerns. In our mitigation, we propose to strengthen the access control of the HCE-based service, aiming to provide our best practice of reducing similar concerns on other 3rd party service providers.

5.5.1 Adversary Model

Fundamentally, HCE security foundation is much weaker than eSE solution. The financial transaction security is built upon Linux kernel protection, which is not enough for this industry. Therefore, the attackers are willing to take the cost of breaking a system, gaining the more valuable and straight-forward financial benefits as return. As a result, the

attackers may leverage the CVEs in the wild to attack the 3rd party service provider (a.k.a HCE service in this scenario).

5.5.2 Design Analysis

From the security perspective, a Root of Trust (RoT) is needed out of the existing host OS. As mentioned in Section 5.5.1, the HCE security build-up the host OS system security is useless if the attackers take the cost to attack the system. Therefore, a more trusted layer out of the host OS is required to serve as a Trusted Computing Base (TCB) in this context. Lots of candidates can fulfill the requirements like virtualization, Secure Enclave, TrustZone, Intel SGX and secure element. Here, we take TrustZone (TZ) as the RoT at Exception Level 3 (EL3) with the following considerations: first, TZ runs fast and secure. TZ OS shares the CPU processors with host OS, but runs in two different modes: secure world and non-secure world. TZ can control buses non-preemptively by controlling a non-secure (NS) bit of switching the two modes. Second, TZ is largely enabled on mainstream mobile devices mainstream ARM processors like Snapdragon, Exynos, and Kirin.

From the implementation perspective, there are challenges of developing mitigation. First, the implementation should be verifiable in some way. Second, it should still make the HCE model easy-to-use with limited impact to wallet apps and the industry. Third, it should not impact the established user experience and payment flow so that the transit does not introduce much burden to the users.

5.5.3 Design

To reinforce the security of HCE-based service, we develop a TrustZone-based RoT for HCE-based digital wallets. As the uniqueness of the 3rd party service provider model, the only legitimate service (NFC service and its controller) is secured with published certificate serving as trusted domain in a trustlet of TrustZone OS. The HCE wallet, therefore, is able to verify the legitimacy of the service and set up an encrypted channel (mimicking the wired channel in SE) to exchange APDUs. In this case, the access of the 3rd party service provider is controlled by the 3rd party and only the service with trusted domains are allowed. This work is based on a previous work [97], which has been patented and granted.

Building The Root Of Trust

As mentioned above, we take advantage of TrustZone technology to build the RoT.

The high-level OS is lack of a RoT for 3rd party to trust. As a 3rd party app, the digital wallet is not capable of distinguishing the trustworthiness of the caller. A trusted agency is required. Such trusted agency secure the NFC service as well as its NFC controller to ensure the Android system is capable to secure the channel between the POS and the wallet. To achieve this, two Trusted Applications (TAs) are built and the two (crypto and NFCC) TAs collaboratively work together in secure world to ensure the end-to-end security between the POS and the wallet. On one hand, crypto TA One stores the certificate of the NFC service, making it recognizable by 3rd party providers. This TA also build a secure channel with cryptography method. In the end, the 3rd party provider and the crypto TA can communicate with an encrypted channel without being sniffed or

hijacked. On the other hand, the NFCC TA builds a minimum NFCC driver to drive the IO of NFCC. The NFCC TA has the priority to read and write the NFCC hardware in a non-preemptive way. By doing so, the APDU data can be securely send and receive between secure world and the POS via NFCC without interruption. In addition, the two TA can communicate and recognize with each other in secure world. So far, the RoT is set up as the foundation of the NFC service and NFCC for 3rd party HCE services. Such design not just works for system service who is associated with hardware like NFC, but also applicable to those with virtual hardware like input methods. For those without physical or virtual hardware, hardware TA is not needed because the end-to-end access is from the system service to 3rd party provider only.

Building The Access To The 3rd Party Service Provider

With the RoT, the access to 3rd party needs to be built. As the 3rd party service provider like HCE-based wallet can trust the RoT, the 3rd party HCE service is able to tell the legitimate system service associated with the RoT. For the access of a provider, two things need to consider: the authorization and the I/O actions including read and write.

For the initial communication setup, an authorization is required. The binding requester initiates the binding request, then the HCE service of the digital wallet confirms the request. After that, the two party starts the authorization following standard RSA-signatures. In this authentication process, the HCE service is required to verify the signature of the request, and it is optional for the requester to verify the HCE service. The RoT provides an API for 3rd party's verification. Other verification method can also work

but not in our design. For example, if the TZ vendor is able to use a public signed certificate, then its signature is verifiable with the help of the ROOT CA. This might be favorable if the HCE service would like to setup its end-to-end communication channel from its cloud instead of the local HCE service. After the HCE service verifies the signature successfully, it can allow the binding request. The authorization process is completed and the binding from the requester to the HCE service is setup.

A TLS-like encrypted channel is used to substitute an end-to-end wired connection between the contactless front-end and computing logic (like SE or HCE). After the binding is completed, the two parties exchange keys following PKI infrastructure and establish an encrypted channel. When calling each other through the interfaces, the two parties encrypt and decrypt the payloads with their cipher. TLS can provide privacy and data integrity between the two parties. No one else is able to eavesdrop messages. Also like wired connection, attackers will be notice if they forcibly interrupt the connection. By far, the end-to-end message exchange is setup between crypto TA and the HCE service. The crypto TA decrypts messages and forward the plain payloads to NFCC TA driver and route the message out from secure world to the hardware (NFCC) directly. Since the secure world rules the processor over the normal world, it possesses the hardware with higher privilege and normal world remains in the blocked state until the secure world finishes, exits voluntarily and switches the processor context to normal world state.

5.5.4 Implementation

We implemented a proof of concept based on the above design and name it as TzNfc. TzNfc was implemented on a Samsung Galaxy S6 with the SOC processor Exynos 7420 chipset and 2 Gb memory. The host OS is Android M OS and the TEE is built with Trustonic TrustZone OS. On this Galaxy S6, the NFC chip is N5DDPS2 made by Samsung.

TA Implementation Considerations

The trustlets are sometimes called trusted application (TA), and the two names are used interchangeably. We build the two TAs with small footprints to make them be programmed verifiably. The NFC TA is implemented as a TA driver to control the NFC controller (NFCC) from TEE environment directly while the crypto TA mainly works as an end point talking to the outside of TEE. The NFCC TA driver is mainly responsible for transmitting data from TEE to NFCC so that it can controls the data transmission between NFCC and POS. While the crypto TA owns the certificate and it facilitates the wallet to verify signature and set up an encrypted message channel between the RoT and the wallet. In the following of this Section, we illustrates how we implement the two TAs in detail.

Access NFCC from TrustZone

Access NFCC from TrustZone requires a TrustZone enabled bus is connected to the NFCC. Depending on the hardware solutions, NFCC can be physically connected to the SOC via SPI, I2C, UART or even USB buses. Our S6 uses I2C bus to connect SOC with NFCC. In fact, not all the buses are enabled for TrustZone to access. We investigated S6's

port of I2C bus connecting to NFC controller and confirmed that it can also be accessed in CPU's monitoring mode. That means this port is enabled for TrustZone TA to access even though it is not the case by default. In our implementation, we developed a I2C driver to have serialized I/O access to the above identified port based on the specs of the NFCC, and were able to drive data transmission successfully between the NFCC and a POS.

NFC Driver in NFCC TA

One of the key principle is NFCC TA Driver should be minimized. Porting/developing irrelevant code in TA can dramatically increase the attack surface and engineering mistakes. As the core of TzNfc, NFCC TA Driver should be formally or at least manually verified. Essentially, there is no need to re-construct the whole NFC driver inside TZ. NFC driver maintains the basic state of NFC controller and provides the interface to user space. NFCC talks to the NFC service with NFC Controller Interface (NCI) messages and NCI messages are sent through a Hardware Abstract Layer (HAL). The HAL defines a relatively standard interface between the NFC driver and the kernel and divides a NFC driver into two parts: hardware related and non-hardware related. To make the TA driver verifiable and portable, we only implement the four basic IO operations (read, write, open and close) in TA driver, leaving the remaining hardware irrelevant logic in Android. Such decision cannot introduce any security concerns, because the application-level APDU messages are encrypted. Once messages are altered, the TA driver can detect it and drop the session. In addition, since only the basic IO operations are implemented in the TA driver, it saves time to port from one NFC chip set to another if needed.

Key Exchange and Authorization

Key exchange and the proper authorization from the wallet is the key prerequisite for the NFC service to successfully bind to the HCE service of the wallet. Such key exchange is integrated into the wallet selection phase. When a user select a wallet or a default wallet was selected, key exchange was triggered. During the key exchange, a preloaded certificate served as the trusted identity was exchanged to the wallet. The wallet implements its own authorization logic to verify the certificate. Such authorization logic can be provided as SDK for 3rd party wallets, but it is out the scope of our POC. For all the opened API on the crypto TA, the requester access the interface through a Secure Monitor Call (SMC) to execute the TA in secure world. Security checks are implemented for system access only APIs.

The encrypted channel

After the key exchange, the wallet and crypto TA (on behave of NFC service) verify each other and set up the cipher and the key for messages encryption later. In the POC, the encryption algorithm is AES-GCM, but not everything in a message is encrypted.

The NFCC encapsulates the message into a NCI message (defined in ISO 7816-4). A common NCI message format (shown in Figure 5.4) can be one of the four types: Data packet, command message, response message and notification message. Although the APDU message is the payload of command and response messages, NFCC is also used for other purposes, e.g. Android's NFC can run different modes like p2p, read/write or HCE. Blindly encrypting the NCI message will make greatly impact other functionalities in

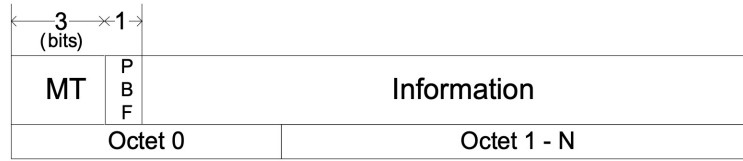


Fig. 5.4.: NCI Core Packet Format

Android, and most important, the NFC service will have no idea of handling a NCI message. To solve this problem, we only encrypt the APDU payloads of the NCI messages, and indicate NFC service if the APDU payload is encrypted. A sample NCI message is depicted in Figure 5.4 from NXP [98]. The MT is a 3 bit field indicating message type, which can represent 8 types but only 3 types are used. We take one Reserved for Future Use (RFU) value to represent the encrypted data type.

For encrypted message, NFC service will route to the crypto TA and send data out of device from TZ directly. For the others, NFC service will handle it as usual, making sure other NFC related functionalities run without impact.

5.5.5 Evaluation

We evaluate the POC from both the effectiveness and performance perspectives. For the effectiveness, we focus on our system implementation. The authorization is implemented by the wallets so we are unable to evaluate in POC.

Effectiveness Experiment

In our implementation, we wrote a HCE-based wallet and implemented the authorization as instructed.

TA implementation effectiveness To verify if the NFCC TA driver can take control of I2C bus exclusively, we removed the mutex on hardware access, and tried to access NFCC from kernel when NFCC TA was accessing I2C bus. Our results demonstrated that the kernel was not able to obtain the handler of the NFCC therefore was not able to access NFCC as planned. When trying to access TAs from other TAs or privileged host services, both TAs refused the unexpected API accesses.

encrypted channel effectiveness To check if the encrypted channel is tamper-resistant, we hijacked the session and distort the original encrypted messages and check if the malicious behavior was detected. Our results showed that the distorted message was able to be detected by the crypto TA and the session was dropped by the NFC service consequently.

Performance

To measure the overhead, we perform multiple tests on TzNfc enabled devices to collect the transaction time of each request and response APDUs and the whole session. We use a hardware contactless card test tool called SmartWave Box made by UL. This tool has a hardware sniffer and can sniff the transaction data with timestamp by detecting the radio wave of a transaction.

As a user, we did not sense much different payment experience. For the tests on transactions of Visa and MasterCard tokens, we got similar results that the overall transaction increases around 100ms on average. By further analyzing the result, we found that the main overhead was divided into two parts. First, we found that the main overhead

was from the encryption and decryption processes, which introduced around 200ms per call. Since the APDU message length is similar, there is no obvious time difference for the encryption and decryption of all kinds of APDU messages. Half of the overhead was from the wallets by design, and the other half was from the crypto TA. We did not pay special attention to the performance improvement but there was room for it if it is commercialized. Second, the SMC calls (switching from normal world to the crypto TA), introduces around 20ms per call. In addition, the NFCC TA driver performed pretty fast compared to the kernel driver, and there was no noticeable difference between the two. Based on our observation, card brands may affect the performance. For example, MasterCard transaction's overhead is a little bigger than Visa's. That is because MasterCard, in general, has one more round of request/response APDU communication compared to Visa in one transaction session.

5.6 Related Work

5.6.1 Attacks and Solutions on NFC and EMV protocols

Previous attacks focus on hardware exploitation and break the EMV protocol. A collection of prior arts [99, 100] have discussed the security concerns of contactless interface. Contactless interface is known to be vulnerable to relay attack known as a type of man-in-the-middle(MITM) attack. Some previous work [101–103] discussed about the flaw of EMV protocol, known as mafia attack. Mafia attack assume that the POS that the customer taps phone to is malicious and routed data to the POS. All of the above work requires the customer to actually tap the smart card/mobile phones to the POS to steal

the payment secret. The consequence of the attack is to make a transaction different from what the consumer see. In real world, this type of attack can easily be identified because the malicious POS can be identified by the customer or card issuer's fraud detection.

To solve each of the above mentioned attacks, several solutions have been proposed. The pre-play attack, as a typical replay attack, has been widely studied. In payment industry, this has been briefly resolved by integrating nonce generation into the application layer protocol. There are some popular approaches to mitigate mafia attack. For example, MITM attacks are usually time sensitive, the extra man-in-the-middle data transmission may introduce some time delay. One of the popular approaches of resolving distance bounding is [104].

5.6.2 Attacks and Protections in Android Payment System

However, using a mobile wallet is not without risks [105–108]. Many researchers have started to propose various security mechanism to secure the android payment transaction at different levels. At the application level, authentication approaches are most common approaches to validate the legitimacy of users [109]. The security of payment transaction usually adopts the cryptography to prevent relay attacks [108]. At the transaction level, many approaches adopt tokens or cryptograms to authorize the payment transaction [105], and develop various secure SSL protocols to guarantee the confidentiality of the transaction. At the hardware level, Secure Element (SE), a tamper-resistant chip with a secure microcontroller, is used to securely store confidential and cryptographic data for the payment transaction. Some of the previous proposed their concerns to the security of HCE

architecture, while we categorize the HCE problem as 3rd party service provider problem and provide concrete real world attacks to further demonstrate the severity of this design concern. Furthermore, different to the previous solutions, we propose a general solution, based on Trusted Execution Environment (TEE), to ensure the trusted and secure execution environment by offering a higher level of functionality than SE alone [110, 111] in Android.

6. SUMMARY

This thesis addresses the security concerns at component level in Android devices by developing a component context-awareness solution. Our layered, system-wide approach consists of three key work: Flasa, Compac and TzNfc. Flasa controls inter-component interactions and database's access at component level, Compac controls component's access to framework and kernel level resources, and TzNfc regulates access to 3rd party service providers.

Through extensive evaluation, we demonstrate that our solution effectively recognizes component contexts and controls access to critical resources, thereby enhancing the overall security of Android devices. By complementing the existing Android security model, our proposed solution raises the security standard in mobile devices, ultimately benefiting users, developers and the broader mobile ecosystem.

LIST OF REFERENCES

LIST OF REFERENCES

- [1] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, “Addroid: Privilege separation for applications and advertisers in android,” in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, AsiaCCS ’12, 2012.
- [2] S. Shekhar, M. Dietz, and D. S. Wallach, “Adsplit: separating smartphone advertising from applications,” in *Proceedings of the 21st USENIX conference on Security symposium*, USENIX Security ’12, (Berkeley, CA, USA), pp. 28–28, USENIX Association, 2012.
- [3] X. Zhang, A. Ahlawat, and W. Du, “AFrame: Isolating Advertisements from Mobile Applications in Android,” in *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC)*, (New Orleans, Louisiana, USA), December 9-13 2013.
- [4] A. E. M. Dawoud and S. Bugiel, “Droidcap: Os support for capability-based permissions in android,” *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.
- [5] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. von Styp-Rekowsky, “Boxify: Full-fledged app sandboxing for stock android,” in *24th USENIX Security Symposium (USENIX Security 15)*, pp. 691–706, 2015.
- [6] 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), p. 165–176, Association for Computing Machinery, 2014.
- [7] E. Athanasopoulos, V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, “Naclroid: Native code isolation for android applications,” in *Computer Security – ESORICS 2016* (I. Askoxylakis, S. Ioannidis, S. Katsikas, and C. Meadows, eds.), (Cham), pp. 422–439, Springer International Publishing, 2016.
- [8] J. Huang, O. Schranz, S. Bugiel, and M. Backes, “The art of app compartmentalization: Compiler-based library privilege separation on stock android,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1037–1049, 2017.
- [9] 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), p. 173–187, Association for Computing Machinery, 2011.
- [10] “Android virtualization framework (avf) overview.”
<https://source.android.com/docs/core/virtualization>.

- [11] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen, “I-arm-droid: A rewriting framework for in-app reference monitors for android applications,” in *Proceedings of the Workshop on Mobile Security Technologies*, MOST ’12, (San Francisco, CA), May 2012.
- [12] N. Reddy, J. Jeon, J. Vaughan, T. Millstein, and J. Foster, 01 2015.
- [13] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein, “Dr. android and mr. hide: fine-grained permissions in android applications,” in *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM ’12, (New York, NY, USA), pp. 3–14, ACM, 2012.
- [14] 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.
- [15] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon, M. Nystrom, D. Robinson, R. Spiger, S. Thom, and D. Wooten, “Ftpm: A software-only implementation of a tpm chip,” in *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC’16, (USA), p. 841–856, USENIX Association, 2016.
- [16] J. Winter, “Trusted computing building blocks for embedded linux-based arm trustzone platforms,” in *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing*, STC ’08, (New York, NY, USA), p. 21–30, Association for Computing Machinery, 2008.
- [17] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, “Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS ’14, (New York, NY, USA), p. 90–102, Association for Computing Machinery, 2014.
- [18] D. Chakraborty, L. Hanzlik, and S. Bugiel, “Simtpm: User-centric tpm for mobile devices,” in *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC’19, (USA), p. 533–550, USENIX Association, 2019.
- [19] K. Ying, A. Ahlawat, B. Alsharifi, Y. Jiang, P. Thavai, and W. Du, “Truz-droid: Integrating trustzone with mobile operating system,” in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys ’18, (New York, NY, USA), p. 14–27, Association for Computing Machinery, 2018.
- [20] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O’Keeffe, M. L. Stillwell, D. Goltzsche, D. Eysers, R. Kapitza, P. Pietzuch, and C. Fetzer, “Scone: Secure linux containers with intel sgx,” in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI’16, (USA), p. 689–703, USENIX Association, 2016.
- [21] K. A. Küçük, A. Pavard, A. Martin, N. Asokan, A. Simpson, and R. Ankele, “Exploring the use of intel sgx for secure many-party applications,” in *Proceedings of the 1st Workshop on System Software for Trusted Execution*, SysTEX ’16, (New York, NY, USA), Association for Computing Machinery, 2016.

- [22] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzner, P. Pietzuch, and R. Kapitza, “Securekeeper: Confidential zookeeper using intel sgx,” in *Proceedings of the 17th International Middleware Conference*, Middleware ’16, (New York, NY, USA), Association for Computing Machinery, 2016.
- [23] S. Smalley and R. Craig, “Security enhanced (se) android: Bringing flexible mac to android,” in *Proceedings of 20th Annual Network & Distributed System Security Symposium*, NDSS ’13, The Internet Society, Feb 2013.
- [24] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi, “Xmandroid: A new android evolution to mitigate privilege escalation attacks,” Technical Report TR-2011-04, Technische Universität Darmstadt, Apr 2011.
- [25] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastri, “Practical and lightweight domain isolation on android,” in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pp. 51–62, 2011.
- [26] S. Bugiel, S. Heuser, and A.-R. Sadeghi, “Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies,” in *Proceedings of the 22nd USENIX conference on Security symposium*, USENIX Security ’13, USENIX, 2013.
- [27] M. Backes, S. Bugiel, S. Gerling, and P. von Styp-Rekowsky, “Android security framework: Extensible multi-layered access control on android,” in *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC ’14, (New York, NY, USA), p. 46–55, Association for Computing Machinery, 2014.
- [28] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau, “The flask security architecture: system support for diverse security policies,” in *Proceedings of the 8th conference on USENIX Security Symposium*, USENIX Security ’99, (Berkeley, CA, USA), pp. 11–11, USENIX Association, 1999.
- [29] 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*, AsiaCCS ’10, (New York, NY, USA), pp. 328–332, ACM, 2010.
- [30] M. Conti, V. T. N. Nguyen, and B. Crispo, “Crepe: context-related policy enforcement for android,” in *Proceedings of the 13th international conference on Information security*, ISC ’10, (Berlin, Heidelberg), pp. 331–345, Springer-Verlag, 2011.
- [31] 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*, ACSAC ’09, (Washington, DC, USA), pp. 340–349, IEEE Computer Society, 2009.
- [32] 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.
- [33] M. Ongtang, K. Butler, and P. McDaniel, “Porscha: policy oriented secure content handling in android,” in *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC ’10, (New York, NY, USA), pp. 221–230, ACM, 2010.

- [34] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, “Quire: lightweight provenance for smart phone operating systems,” in *Proceedings of the 20th USENIX conference on Security symposium*, USENIX Security ’11, (Berkeley, CA, USA), pp. 23–23, USENIX Association, 2011.
- [35] 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*, USENIX Security ’11, (Berkeley, CA, USA), pp. 22–22, USENIX Association, 2011.
- [36] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry, “Practical and lightweight domain isolation on android,” in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM ’11, (New York, NY, USA), pp. 51–62, ACM, 2011.
- [37] 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.
- [38] Y. Zhou and X. Jiang, “Detecting passive content leaks and pollution in android applications,” in *Proceedings of 20th Annual Network & Distributed System Security Symposium*, NDSS ’13, pp. 1–6, The Internet Society, 2013.
- [39] X. Jiang, “New rogue android app – roguesppush – found in alternative android markets.” <http://www.csc.ncsu.edu/faculty/jiang/RogueSPPush>.
- [40] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry, “Towards taming privilege-escalation attacks on android,” in *Proceedings of 19th Annual Network & Distributed System Security Symposium*, NDSS ’12, The Internet Society, Feb 2012.
- [41] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, “Systematic detection of capability leaks in stock android smartphones,” in *Proceedings of 19th Annual Network & Distributed System Security Symposium*, NDSS ’12, 2012.
- [42] “Application fundamentals.” <http://developer.android.com/guide/components/fundamentals.html>.
- [43] A. K. Maji, F. A. Arshad, S. Bagchi, and J. S. Rellermeyer, “An empirical study of the robustness of inter-component communication in android,” in *Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN ’12, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2012.
- [44] “Android open source project. android security overview.” <http://source.android.com/tech/security>.
- [45] “The androidmanifest.xml file.” <http://developer.android.com/guide/topics/manifest/manifest-intro.html>.
- [46] 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.

- [47] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell, "The inevitability of failure: The flawed assumption of security in modern computing environments," in *Proceedings of the 21st National Information Systems Security Conference*, NISS '98, pp. 303–314, 1998.
- [48] "The androidmanifest.xml file." <http://developer.android.com/guide/topics/manifest/manifest-intro.html>.
- [49] "Sqlite home page." <http://www.sqlite.org>.
- [50] L. Vogel, "Android development tutorial." <http://www.vogella.com/articles/Android/article.html>.
- [51] J. P. Anderson, "Computer security technology planning study," Technical Report TR-2011-04, DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS HQ ELECTRONIC SYSTEMS DIVISION (AFSC), Apr 1972.
- [52] T. Jeager, *Operating System Security*. Synthesis Lectures on Information Security, Privacy, and Trust, <http://www.morganclaypool.com/doi/abs/10.2200/S00126ED1V01Y200808SPT001>, 2008.
- [53] "Setools - policy analysis tools for selinux," <http://oss.tresys.com/projects/setools>.
- [54] "Talking about android message queue." <http://cphacker0901.wordpress.com/1900/12/22/talking-about-android-message-queue>.
- [55] 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 Symposium on Information, Computer and Communications Security*, AsiaCCS '13, 2013.
- [56] "Samples — android developers." <http://developer.android.com/tools/samples/index.html>.
- [57] "Android.jsmshider — symantec." http://www.symantec.com/security_response/writeup.jsp?docid=2011-062114-0857-99.
- [58] "Android/hipposms.a — virus profile & definition — mcafee inc.." <http://home.mcafee.com/virusinfo/virusprofile.aspx?key=544065>.
- [59] "Antutu benchmark — know your android better." <http://www.antutu.com>.
- [60] "R1 benchmark: Sqlite - android apps on google play." <https://play.google.com/store/apps/details?id=com.redlicense.benchmark.sqlite&hl=en>.
- [61] R. Cox, "Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby, ...)." <http://swtch.com/~rsc/regexp/regexp1.html>.
- [62] "Sqlitedatabase — android developers." <http://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html>.
- [63] T. Harada, T. Horie, and K. Tanaka, "Task oriented management obviates your onus on linux," in *Linux Conference*, 2004.
- [64] "Seforandroid - selinux wiki." <http://http://selinuxproject.org/page/SEAndroid>.

- [65] S. Bugiel, S. Heuser, and A.-R. Sadeghi, “Towards a framework for android security modules: Extending se android type enforcement to android middleware,” Technical Report TR-2012-12, Technische Universität Darmstadt, Apr 2012.
- [66] R. Xu, H. Saïdi, and R. Anderson, “Aurasium: practical policy enforcement for android applications,” in *Proceedings of the 21st USENIX conference on Security symposium*, USENIX Security ’12, (Berkeley, CA, USA), pp. 27–27, USENIX Association, 2012.
- [67] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, “Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones,” in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI ’10, (Berkeley, CA, USA), pp. 1–6, USENIX Association, 2010.
- [68] 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.
- [69] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan, “User-driven access control: Rethinking permission granting in modern operating systems,” in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, S&P ’12, (Washington, DC, USA), pp. 224–238, IEEE Computer Society, 2012.
- [70] K. Z. Chen, N. Johnson, V. D’Silva, S. Dai, K. MacNamara, T. Magrino, E. X. Wu, M. Rinard, and D. Song, “Contextual policy enforcement in android applications with permission event graphs,” in *Proceedings of the 20th Annual Network & Distributed System Security Symposium*, NDSS ’13, (San Diego, USA), February 2013.
- [71] “Android marks fourth anniversary since launch with 75.0third quarter, according to idc.” <https://www.idc.com/getdoc.jsp?containerId=prUS23771812>.
- [72] Y. Zhou and X. Jiang, “Dissecting android malware: Characterization and evolution,” in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, S&P ’12, (Washington, DC, USA), pp. 95–109, IEEE Computer Society, 2012.
- [73] “Android ndk.” <http://developer.android.com/tools/sdk/ndk/index.html>.
- [74] 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 *Proceedings of 19th Annual Network & Distributed System Security Symposium*, NDSS ’12, The Internet Society, Feb 2012.
- [75] 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*, S&P ’09, (Washington, DC, USA), pp. 79–93, IEEE Computer Society, 2009.
- [76] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, “Efficient software-based fault isolation,” in *Proceedings of the 14th ACM symposium on Operating systems principles*, SOSP ’93, (New York, NY, USA), pp. 203–216, ACM, 1993.
- [77] M. Sun and G. Tan, “Jvm-portable sandboxing of java’s native libraries,” in *Proceedings of the 17th European Symposium on Research in Computer Security*, vol. 7459 of *ESORICS ’12*, pp. 842–858, 2012.

- [78] J. Siefers, G. Tan, and G. Morrisett, “Robusta: taming the native beast of the jvm,” in *Proceedings of the 17th ACM conference on Computer and communications security*, CCS ’10, (New York, NY, USA), pp. 201–211, ACM, 2010.
- [79] “The security manager.” <http://docs.oracle.com/javase/tutorial/essential/environment/security.html>.
- [80] 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.
- [81] W. Enck, D. Ocateau, P. McDaniel, and S. Chaudhuri, “A study of android application security,” in *Proceedings of the 20th USENIX conference on Security symposium*, USENIX Security ’11, (Berkeley, CA, USA), pp. 21–21, USENIX Association, 2011.
- [82] L. K. Yan and H. Yin, “Droidscape: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis,” in *Proceedings of the 21st USENIX conference on Security symposium*, USENIX Security ’12, (Berkeley, CA, USA), pp. 29–29, USENIX Association, 2012.
- [83] L. Gong, “Java security architecture (jdk1.2).” <http://docs.oracle.com/javase/1.4.2/docs/guide/security/spec/security-spec.doc.html>, 1998.
- [84] “Android malware genome project.” <http://www.malgenomeproject.org>.
- [85] “Package java.lang.instrument.” <http://docs.oracle.com/javase/6/docs/api/java/lang/instrument/package-summary.html>.
- [86] “Testing fundamentals.” http://developer.android.com/tools/testing/testing_android.html.
- [87] “Angry birds.” <http://www.angrybirds.com>.
- [88] “Phonegap.” <http://phonegap.com>.
- [89] 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.
- [90] “Abusing webview javascript bridges.” <http://50.56.33.56/blog/?m=201212>.
- [91] “Linpack for android - android apps on google play.” <https://play.google.com/store/apps/details?id=com.greenecomputing.linpack&hl=en>.
- [92] A. Bousquet, J. Briffaut, L. Clévy, C. Toinard, and B. Venelle, “Mandatory Access Control for the Android Dalvik Virtual Machine,” in *Workshop on Embedded Self-Organizing Systems (ESOS)*, 2013.
- [93] A. Boxall, “androidbank.”
- [94] Statista, “androidpaymarket.” <https://www.statista.com/statistics/244475/proximity-mobile-payment-transaction-value-in-the-united-states/>.
- [95] “Capital one wallet.” <https://www.capitalone.com/applications/mobile/wallet/>.

- [96] Statista, “Wells fargo wallet.”
<https://www.wellsfargo.com/mobile-payments/wells-fargo-wallet/>.
- [97] P. P. P. N. Yifei Wang, An Liu, “System and method for secure transactions with a trusted execution environment (tee),” 2018. US Patent 16/122,705.
- [98] NXP, *UM10819 PN7120 User Manual*, 2017.
- [99] Y. Desmedt, C. Goutier, and S. Bengio, “Special uses and sbuses of the fiat-shamir passport protocol,” in *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, CRYPTO ’87, (London, UK, UK), pp. 21–39, Springer-Verlag, 1988.
- [100] S. Akter, S. Chellappan, T. Chakraborty, T. A. Khan, A. Rahman, and A. B. M. Alim Al Islam, “Man-in-the-middle attack on contactless payment over nfc communications: Design, implementation, experiments and detection,” *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 6, pp. 3012–3023, 2021.
- [101] M. Bond, O. Choudary, S. J. Murdoch, S. Skorobogatov, and R. Anderson, “Chip and skim: Cloning emv cards with the pre-play attack,” in *2014 IEEE Symposium on Security and Privacy*, pp. 49–64, 2014.
- [102] M. Roland and J. Langer, “Cloning credit cards: A combined pre-play and downgrade attack on emv contactless.,” in *WOOT*, 2013.
- [103] M. Bond, M. O. Choudary, S. J. Murdoch, S. Skorobogatov, and R. Anderson, “Be prepared: The emv preplay attack,” *IEEE Security Privacy*, vol. 13, no. 2, pp. 56–64, 2015.
- [104] S. Drimer and S. J. Murdoch, “Keep your enemies close: Distance bounding against smartcard relay attacks,” in *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, SS’07, (Berkeley, CA, USA), pp. 7:1–7:16, USENIX Association, 2007.
- [105] M. Roland, “Software card emulation in nfc-enabled mobile phones: great advantage or security nightmare,” in *Fourth International Workshop on Security and Privacy in Spontaneous Interaction and Mobile Phone Use*, pp. 1–6, 2012.
- [106] L. Francis, G. Hancke, K. Mayes, and K. Markantonakis, “Practical nfc peer-to-peer relay attack using mobile phones,” in *International Workshop on Radio Frequency Identification: Security and Privacy Issues*, pp. 35–49, Springer, 2010.
- [107] Z. Kfir and A. Wool, “Picking virtual pockets using relay attacks on contactless smartcard,” in *Security and Privacy for Emerging Areas in Communications Networks, 2005. SecureComm 2005. First International Conference on*, pp. 47–58, 2005.
- [108] G. P. Hancke, “A practical relay attack on iso 14443 proximity cards,” *Technical report, University of Cambridge Computer Laboratory*, vol. 59, pp. 382–385, 2005.
- [109] J.-Y. Hu, C.-C. Sueng, W.-H. Liao, and C. C. Ho, “Android-based mobile payment service protected by 3-factor authentication and virtual private ad hoc networking,” in *Computing, Communications and Applications Conference (ComComAp)*, 2012, pp. 111–116, 2012.

- [110] A. Vasudevan, E. Owusu, Z. Zhou, J. Newsome, and J. M. McCune, “Trustworthy execution on mobile devices: What security properties can my mobile platform give me?,” in *International Conference on Trust and Trustworthy Computing*, pp. 159–178, 2012.
- [111] P. Urien and X. Aghina, “Secure mobile payments based on cloud services: Concepts and experiments,” in *Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing (HPSC), and IEEE International Conference on Intelligent Data and Security (IDS), 2016 IEEE 2nd International Conference on*, pp. 333–338, 2016.

VITA

VITA

Yifei Wang was born in Zibo, Shandong Province, China. He received his Bachelor of Science degree in Computer Science from Sun Yat-Sen University, China. He received his Master degree in Computer Information Science and Engineering from Syracuse University in May 2011.