View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by ScholarBank@NUS

Mechanisms for Resource Protection on the Android Platform

LI XIAOLEI

(*B.Eng., TSINGHUA UNIVERSITY*)

A THESIS SUBMITTED

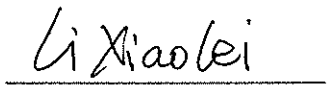
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE

2014

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.



Li Xiaolei

20 September 2014

Acknowledgements

I would like to thank my advisor Professor Zhenkai Liang, for his constant guidance and advice on my varied research interests along my study. He constantly gives me important suggestions and encouragement on both my work and life since the first year of my Ph.D program. With his guidance, I make steady progress and also build up the confidence on my study. Most importantly, he taught me to understand the importance of thinking, which helps me to work for a deep and clear insight towards problems at the very beginning.

I am also indebted to all of the collaborators over the years for their kind help and support. Especially, I would like to thank Kailas Patil, Xinshu Dong, Mingwei Zhang, Aravind Prakash, Guangdong Bai, Hong Hu, Yaoqi Jia, Ting Dai, Behnaz Hassanshahi, Mayank Dhiman, Joseph Hong, and Professors Xuxian Jiang, Heng Yin, Prateek Saxena. I am lucky to collaborate with them on research projects on various topics. They have brilliant suggestions and also work so hard on the projects. I benefit a lot from them when working together with them, not only their enthusiasm on the research work but also their understanding and kindness in the teamwork. I would also like to thank Professors Roland H. C. Yap, Ee-Chien Chang and Tulika Mitra for their kind support and recommendation on my research study. Finally, I would like to thank all my labmates for their kind help on my study and life, especially Utsav Saraf, Sai Sathyanarayan, Bodhisatta Barman Roy, Zheng Leong Chua, Ziqi Yang, Xuhui Liu, Benjamin Thian, Dongyan Zhang, Jiangang Wang, Yue Chen, Yongzheng Wu, Wei Xia, Liming Lu, Jia Xu, Xuejiao Liu, Junjie Jin, Chengfang Fang, Chunwang Zhang, Xiaolu Zhu, Zhaofeng Chen, Hossein Siadati, Deepak Kathayat, Hoon Wei Lim, Loi Luu, Hung Dang, Shweta Shinde, Shruti Tople, Enrico Budianto, Inian Parameshwaran, Pratik Soni. Besides, many friends have brightened my life and encouraged me a lot. I am sincerely grateful for all their kind help and sharing the best memories with me.

Contents

Abstract

As Android devices become increasingly popular worldwide, security issues also become severe. Threats to sensitive resources, such as user privacy violation and premium service abusing, have become a big concern. Even though the Android system applies a permission-based model to regulate the resource access by Android applications (apps), malicious apps still get the chance to abuse the available resources. To address the threats to sensitive resources, in this thesis we propose new frameworks on the Android platform to enhance resource protection for diverse demands.

To mitigate the threats to sensitive system resources (e.g., user contacts, location data) by malicious apps, we propose a virtualization-based framework that provides a sandbox environment for Android resources. It simulates a virtual but consistent view of the sensitive resources. The resource access by an app is confined inside a virtual view. This framework provides transparent data protection with high compatibility with the existing Android apps.

To allow the sensitive data access while ensuring the tight control, we provide a tightly-controlled and resource-constrained environment. Specially, we build our prototype on the ARM TrustZone architecture, which provides a trusted environment with strong security guarantee by the hardware-level protection. It provides a standalone constrained runtime environment which is completely separate with the Android OS.

Finally, to provide more comprehensive understanding about the potential threats to sensitive resources by a given app, we design a scalable static analysis mechanism on how real-world apps utilize sensitive data, specifically, the impact of a set of operations on the sensitive data. With this comprehensive knowledge regarding resource usage, users are enabled to assess potential threats of unknown apps to their sensitive resources and rank them according to usage patterns to sensitive resources.

With the proposed solutions, we are able to reinforce the resource protection on the existing Android platform with different levels of security guarantees.

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Smartphones are evolving into one of the most important computing and communication tools in our daily life. Compared to traditional mobile phones, smartphones have stronger connectivity capability and more advanced hardware sensors. Therefore, with their provided flexibility, mobility and rich functionality, smartphones become a platform that integrates a rich collection of sensitive data (e.g., phone state, location, user contacts, SMS, calendar, external storage) and services (e.g., sending/receiving SMS, making phone call, establishing network connections), which we refer to as *resources* in this thesis. They also allow users to extend their functionality through a rich selection of third-party mobile applications (apps). Most popular mobile platforms, such as Android and iOS, provide app stores that allow third-party apps to be downloaded and installed onto mobile devices. It becomes popular for people to use their smartphones to do social networking, share personal photos and even make online payment transactions, which increases the chance of sensitive data damage or leakage on the mobile platform.

The Android OS [14], released at 2008, keeps increasing in the worldwide smartphone market share. Gartner's statistics [36] shows that Android's market share has increased from 66.4% in 2012 to 84.6% in 2014. Android-based software development also keeps increasing at a fast speed. Statistics from AndroLib [15] shows that the number of available apps in the official Android market (i.e., Google Play Store) has already surpassed 1.2

million in July 2014, and the total number of app downloads has exceeded one billion. Low expense of hardware and rich support of software make Android devices increasingly popular worldwide. However, as the Android platform gains a large user base, it also becomes a favorite target for attackers.

Security Threats on the Android Platform. Figure 1.1 illustrates the overview of the Android ecosystem. Due to the openness of app stores and a large developer community for the Android platform, the security of publicly available apps is hardly guaranteed by app stores. Developers can easily publish vulnerable apps or deliberately design malware, and upload them to app stores. Users store sensitive data into their mobile devices, and also install third-party apps that are downloaded from the Android market to manage those sensitive data. The mobile device provides a platform for loosely-controlled apps to manage user sensitive data, which often exposes the user sensitive resources to untrusted apps.

The threats to sensitive resources are severe in the Android ecosystem. Malicious apps usually harvest sensitive information or abuse sensitive services, which becomes the main threats on the Android platform. According to the Android malware dissection [111], most of them harvest various information from infected devices, such as device ID, location, user contacts, SMS, and then leak them to a third-party remote server through hidden channels. Recent research [37, 56, 113] has revealed that a large portion of apps expose phone states or location information. Some of them also stealthily send SMS to premium numbers, which causes financial loss of victim users. Malicious apps can also capture stealthy audio/video through microphone and camera services [88, 102]. Furthermore, well-known spyware and rootkits, such as Gingerbreak, Android/Multi.dr, HongToutou, DroidDream and DroidKungFu, can gain the root privilege by exploiting OS-level vulnerabilities and thus hold unlimited control over victim devices, raising high threats to sensitive resources. They are likely repacked into new popular legitimate apps to disguise themselves.

Problem Analysis. To regulate the resource access, the Android system provides a

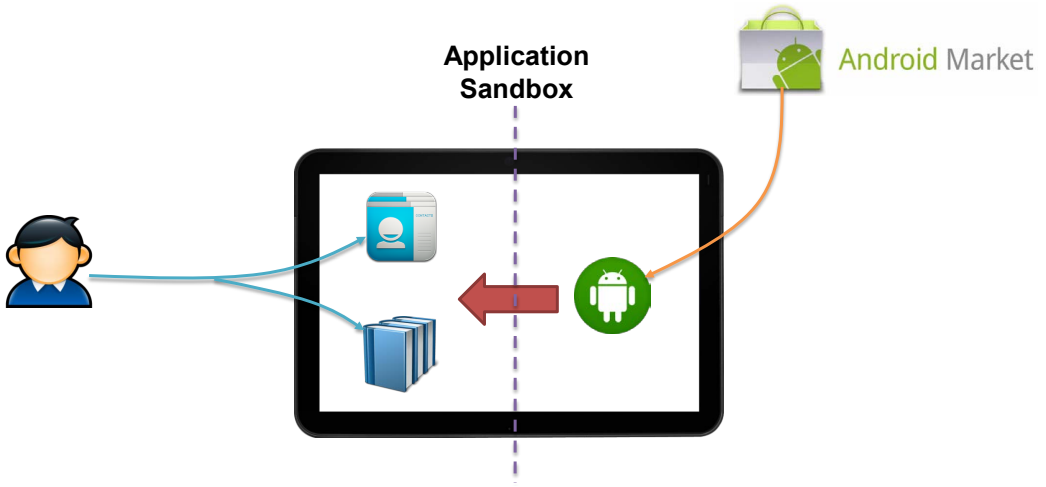


Figure 1.1: User, Device, App Store in the Android Ecosystem

kernel-level resource-centric sandbox environment for restricting the capability of installed apps. The application sandbox environment provides access only to allowed system resources with strict isolation among apps. The access to sensitive resources out of the sandbox, such as user contacts, has to be performed through dedicated protected APIs. These protected APIs are regulated by a permission-based model. An app needs to explicitly request the corresponding permission for accessing a particular resource, and this permission has to be explicitly granted by users at app installation time. For example, in order to access user contacts, the app must specify the `READ_CONTACTS` permission in its configuration file. During app installation, the Android system prompts the list of permissions to users and obtains their permission to install. Although end users seem to be in control to protect their own resources, it is challenging in practice to guarantee resource security.

First, the default protection mechanism relies on users to make a one-for-all decision for all the resources requested by an app, which is not enough to satisfy diverse demands to protect different types of resources. For example, the SD card storage resource provides a general support for storing data shared by all the installed apps. When granting the storage access to one app, we may wish that this app is not going to corrupt other apps' data on the storage. However, for more critical resources, such as credit card number and user credentials, we require a strong protection guarantee that these critical data can only

be accessed by selected users or apps. Therefore, based on the importance of the resources and their usage scenarios, we need diverse system mechanisms to provide different levels of resource protection guarantees.

Second, the permission requests only inform users about what resources one app requires, which is not sufficient for users to assess the potential threats to these granted resources. For example, if one app requests both location permission and network permission, it could divulge users' location data to a malicious tracking system. In order to help users to assess the app threats and make proper security decisions, it is necessary to provide more comprehensive understanding about the internal resource usage inside one app.

By reviewing the current security design on the Android platform from the resource angle, in this thesis, *we propose new effective and practical system mechanisms and analysis techniques to enhance the protection for diverse resources on the Android platform*. It is a big challenge to design practical protection mechanisms for diverse demands, which at the same time balances security and usability.

Resource-centric Enhancement. Many existing work [40, 74, 20, 113, 75, 35, 23, 24] has made efforts to extend the default permission-based protection in Android by either enforcing fine-grained access control or supporting rich-semantic constraints on access. Nevertheless, these solutions either are ad-hoc or increase the complexity of user decisions, resulting in poor usability. It is non-trivial for end users to deal with complex policies and make proper security-related decisions. Therefore, we need new mechanisms to enhance resource protection on the Android platform while still preserving good usability for diverse resources.

For general system resources that are commonly shared by multiple apps in Android devices, such as user contacts, location and external storage, to confine the access by untrusted apps, the view of these resources to mutually untrusted apps should be separated. To be transparent to existing Android apps and thus gain high compatibility, we design a virtualization-based isolation mechanism that only provides a virtual copy of resources

to a particular group of apps. It allows multiple virtual environments to be coexisting but mutually isolated on top of physical resources.

For more critical sensitive resources, such as user credentials and credit card numbers, apps require direct access on them to function properly. Instead of providing a virtual copy of them to apps, we have to unavoidably grant apps the access to the real sensitive data. Therefore, we create a dedicated trusted execution environment that provides stronger protection over these critical data access. It is isolated from the Android OS with hardware-level protection guarantee but supports a primitive set of sensitive data operations.

To assess the threats to sensitive resources from an app, we design a mechanism that analyzes resource usage in Android apps. This approach can be adopted by app stores to rank apps according to their behaviors of accessing resources.

1.1 Thesis Overview

In this thesis, we propose three mechanisms to enhance resource protection on the Android platform to satisfy diverse protection demands for sensitive resources. More specifically, we develop a virtualization-based isolation mechanism to provide transparent protection for resource access, a hardware-level isolation mechanism to provide a tightly-controlled and resource-constrained environment, and a resource-usage-based ranking system for Android apps.

Transparent Protection through Resource Virtualization. To isolate system resources shared among apps, we provide a virtualization-based mechanism to provide a virtual copy of resources to apps. We group the installed apps and provide separate virtual set of resources to each particular group. Apps cannot access the virtual resources that belong to other groups. We reinforce the Android system by adding a new layer between the apps and various types of sensitive data and services. This layer mediates all the sensitive resource access and provides a virtual resource view to apps, such as a virtual SD card and

a virtual user contact database. It is completely transparent to Android apps. Intuitively, we simulate a fresh device environment for running risky apps, so that the potential damage on the sensitive resources by an unknown app is constrained inside its own virtual environment.

Strong Protection with Tightly-controlled Resource Access. For critical resources, such as user credentials, we have to expose them to an app for functionality, instead of a virtual copy. In this case, to prevent arbitrary access, we design a mechanism by taking advantage of a separate trusted environment that ensures tightly-controlled resource access. Inside the trusted environment, we allow the operations on the raw sensitive data but ensure tight control on the accessing authorities and supported operations. To provide a strong protection guarantee, we leverage a hardware-level protection mechanism, the ARM TrustZone architecture. It supports the concept of red/green systems, in which the hardware resources (e.g., memory and storage) are partitioned into a general-purpose untrusted (red) environment and a highly-constrained trusted (green) environment. The red partition with rich hardware resources available (such as memory and storage) is used for running the Android OS, while the green partition with constrained resources is used for running our trusted environment. Our trusted environment is completely separate with the Android OS, thus preventing any threats from even a compromised Android OS. Based on this root of trust, our trusted environment leverages cryptographic-based techniques to ensure that only authorized code can operate on the raw sensitive data.

Understand Resource Usage for Threat Assessment. To provide more comprehensive knowledge regarding the resource usage for threat assessment, we propose an analysis mechanism to reveal how apps utilize sensitive data. For example, we should be able to inform users of the difference between an app that sends the raw user location to third parties, and another app that only provides a yes/no answer to whether the user is presently at a certain museum or not. We use a sequence of operations on the sensitive data, named as the resource usage pattern. We build an analysis tool to automatically extract location data usage patterns from real-world Android apps. According to different usage patterns,

we rank the potential risks on location data given an unknown app.

Summary of Contributions. By investigating techniques to protect sensitive resources on the Android platform, this dissertation makes the following contributions.

- For system resources shared by installed apps, we propose a virtualization-based resource protection mechanism to provide a transparent and highly-compatible environment for resource access.
- For critical resources, we reinforce the Android platform with hardware-assisted protection to provide a tightly-controlled trusted environment that supports stronger data protection and feasible data operations.
- We design an analysis mechanism for evaluating real-world apps to provide comprehensive understanding about their location resource usage.

Chapter 2

Background and Literature Review

2.1 Android Infrastructure

The Android software architecutre includes an operating system, a middleware layer, an application framework and applications, illustrated as Figure 2.1. It is a Linux-based mobile platform. The middleware, written in C/C++ and Java, provides access to native libraries and third-party libraries for the upper layer, such as OpenGL and Webkit. For ease of development, Android provides an application framework which provides well-defined interfaces for apps to manage system resources conveniently. Applications, mostly written in Java language (also possibly including native code), run in a separate Android customized Java virtual machine, Dalvik.

Android Application. An Android app is usually packaged into one *apk* format file, an vairant of JAR file. Although apps are developed in Java language, they do not run as Java `.class` format in standard Java virtual machine. Java source code will be firstly compiled into standard Java bytecode, and then optimized to `.dex` format which is the Android-specific bytecode format. The `dex` format is designed to be more memory-efficient than Java standard class file. Then the bytecode is packaged into one apk package with other resource files including the manifest file, UI layout, localization, etc. Android also provides several built-in Android apps, such as email, SMS, browser, contacts and

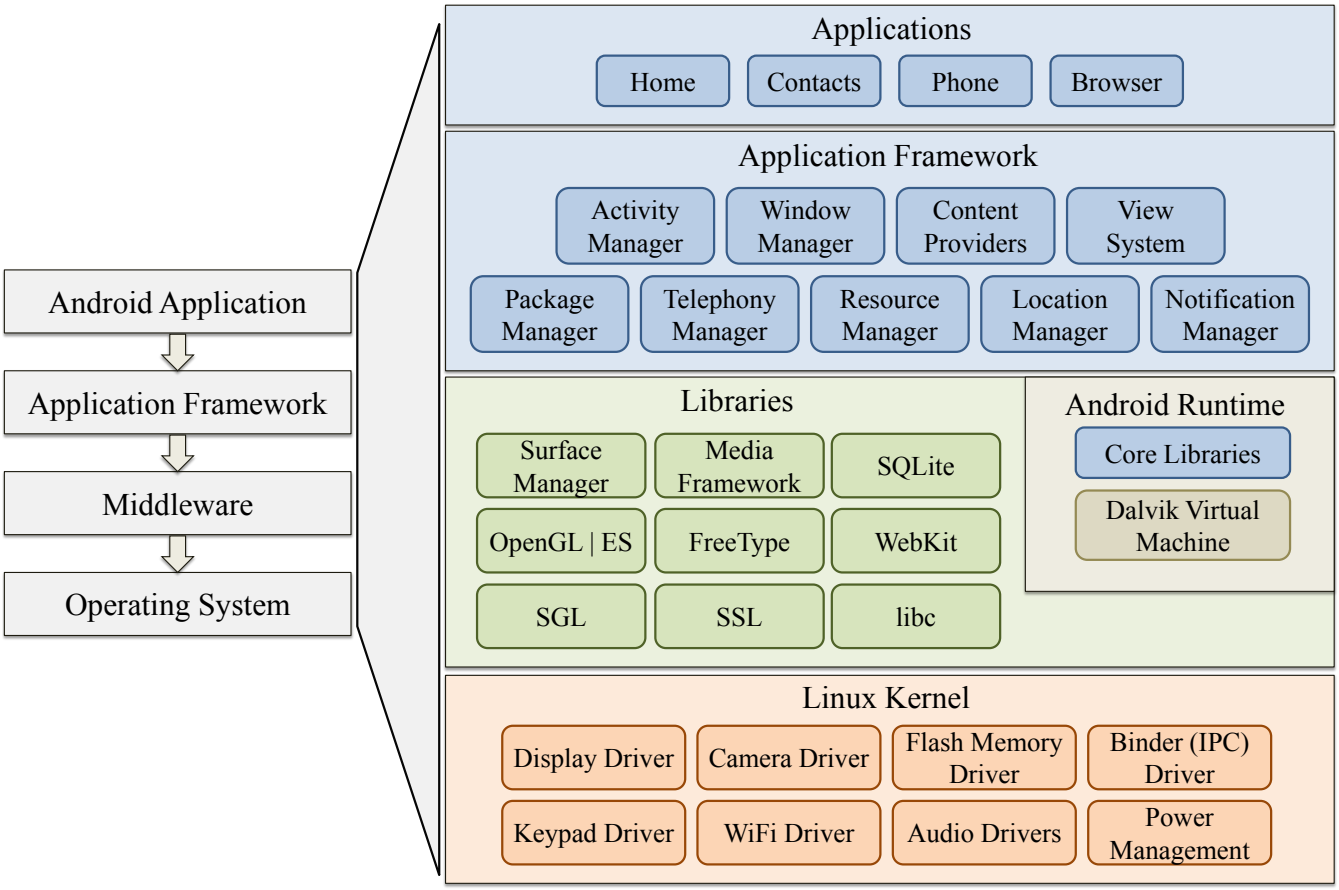


Figure 2.1: Android Software Stack Layout

others.

Application Framework. The application framework layer, written in Java language, simplifies app development and provides well-defined user interfaces for developers to ultilize the underlying functionality. For example, content provider component provides interfaces of accessing contact data from apps. Resource manager provides access to non-code resources such as layout files, and so on. Developers have full access to the framework APIs. They can also publish their own components which other apps may reuse to build rich and innovative apps.

Android Middleware. Apps run on top of the Android middleware that is written in C/C++ and Java language. It provides Java interfaces for apps to directly invoke native system components written in C/C++. It includes libraries that provide various services, such as data storage, screen display, multimedia and web browsing, and also implements

device-specific functions, so that the upper layer does not need to concern variations between various Android devices. Third-party libraries, such as OpenGL, Webkit, can also be loaded to provide rich and convenient integrated functionality. It also contains the Dalvik virtual machine and core Java application libraries. Dalvik is an Android-specific virtual machine. As described above, Android apps are compiled into `dex` bytecode format that will be interpreted in Dalvik VM at runtime, which is a register-based architecture, as opposed to Java VM which is a stack machine. Each app runs in its own Dalvik VM. Core libraries written in Java provide a substantial subset of standard Java packages as well as Android-specific libraries.

Default Protection on the Android Platform. As a Linux-based system. Android sets up a kernel-level application sandbox based on UNIX-style protection mechanisms, such as user separation of processes and file permissions. The Android system assigns a unique user ID to each Android app, and thus the kernel separates apps through standard Linux facilities. By default, an app cannot interact with other apps or access data that belong to other apps, unless through protected Android-specific APIs. These protected APIs are the only way for apps to interact with other apps and access a limited range of system resources. Android applies a permission-based model as a specific security mechanism to restrict apps from accessing protected resources (e.g., user contacts, SMS, location and external storage) through these protected APIs. To access protected resources, each app needs to request the corresponding permissions explicitly during installation. Users have to decide whether they want to trust third-party apps and grant dangerous permissions, such as network permission and contact read/write permission.

However, the permission-based model is not sufficient for resource protection. First, it relies too much on users to make wise security-related decisions. Given a list of permissions, it is not apparent to know whether an app is benign. For example, a combination of `READ_CONTACTS` and `INTERNET` permissions indicate that the app to be installed may read user contacts and then send them to a remote server. Second, its "all-or-none" option is not sufficient. Users can either allow all the permissions requested by a risky app

during the installation, or deny the installation. Though the Android 4.3/4.4.1 releases have a hidden feature, Apps Ops, which allows users to dynamically revoke permissions for an app, a simple permission removal may break app functionality, or even crash the app [59]. Therefore, instead of providing attractive flexibility as expected, it may break apps after permission removal, resulting in bad user experience. This feature also makes the security decisions more complicated for users. Thus Google has completely disabled this feature since the Android 4.4.2 release.

2.2 Literature Review

Existing techniques have been proposed to reinforce the Android software stack from various angles. In this section, we discuss research on resource protection and analysis on the Android platform.

2.2.1 Enhance the Android Permission Model

Android applies a permission-based mechanism to confine the resource access of Android apps. In this mechanism, one app has to request the corresponding permission to access certain sensitive resource. During installation time, the package installer prompts all the permissions required by this app. The Android system only gives users an "all-or-none" choice to grant permissions to an app. When installing an unknown app, users have to either grant all the dangerous permissions or deny the installation. Users cannot selectively grant a subset of permissions requested by one app during installation. After an app gets installed, users cannot later manage a granted permission. Several existing solutions aim to enhance the permission-based protection mechanism by either enforcing more fine-grained access control or supporting rich-semantic constraints on access. Below we discuss these two main categories for enhancing the default permission-based model in Android.

2.2.1.1 Flexible Permission Management

The following solutions propose flexible permission management that allows users to flexibly manage the apps' permissions at any time and even enlarge the options of currently defined permissions.

Kirin [40] modifies the package installer to additionally check whether the permissions requested by the installing app violate a given system-centric policy. The main goal of Kirin is to mitigate malware contained within a single app. For example, to prevent malware from tampering with incoming SMS messages, it defines a security rule that an app must not have `RECEIVE_SMS` and `WRITE_SMS` permission labels, and enforces the policy at the installation time. The expressibility of Kirin is still limited to the existing Android permissions due to the static nature of their enhanced model.

Apex [74] enhances package installer to allow users to permit a subset of requested permissions during the installation. It modifies the Android framework to enforce runtime reference monitor on the permission check. Therefore, due to the flexibility of their dynamic reference monitor framework, it even expands pre-defined Android permissions to support advanced policies, for example, users can not only grant `SEND_SMS` permission as before to allow an app to send SMS, but also can specify the maximum number of SMS messages that can be sent in one day.

TISSA [113] enriches the existing permission-based model towards taming information-stealing problem. In the existing permission-based system, users grant `READ_CONTACTS` permission to a single app to allow its access to the user contacts. TISSA additionally gives users a further option to only let it view a bogus or empty contacts list, instead of the real data. It defines a privacy mode for each app. In this mode, the Android system intercepts the resource access and uses fake sensitive data to prevent untrusted apps from stealing private information.

2.2.1.2 Enhance Constraint on Inter-component Communication (ICC)

Android provides well-defined interfaces for different components to communicate with each other. One component in one app can also interact with components belonging to another app, as long as the permission checking succeeds. Due to this feature, Davi et al. [33] address another weakness of the permission-based model in Android, i.e., the privilege escalation problem (high-privileged apps can process high-privileged tasks requested by other low-privileged ones). Especially, confused deputy attack is a special type of privilege escalation attack where a malicious app abuses the interfaces of a trusted app to perform unauthorized operations. Therefore, a lot of research has been done to strengthen the constraints on the ICC.

Saint [75] implements a reference monitor framework on the ICC and supports fine-grained constraints on both the caller and the callee components at runtime. An ICC is only permitted when the caller and the callee components satisfy user pre-defined conditions, e.g., apps' holding permissions, signatures, release versions and even context-based conditions (such as location and date). App developers have to assign appropriate security policies for each communication interface to specify the properties that the other side (either caller or callee) should hold. It provides a flexible framework for benign security-critical apps to apply strict policies for preventing potential dangerous communication with other untrusted apps. However, considering the practicability, it is non-trivial for developers to consider all security threats into the policies for each individual app.

QUIRE [35] is another Android security extension that prevents confused deputy attack by validating the original app who issues the ICC. It instruments the Android Binder IPC which is a low-level support module in the Android middleware for ICC. QUIRE tracks and records the IPC chain to figure out the original app that starts it. It enables apps to propagate the call-chain context to downstream callees and to authenticate the origin of data that they receive indirectly. Therefore, it allows endpoints that protect sensitive resources to reason about the complete IPC call-chain before granting access to the requesting app. In their design, the additional security context is passed explicitly as an

argument in the IPC which is manageable by apps and enables an app to exercise a privilege if it genuinely wishes so. However, it cannot prevent colluding apps that may drop the caller in IPC chain deliberately and act on their own behalf.

XManDroid [23] is another reference monitor system that extends the Android application framework layer to detect and prevent application-level privilege escalation attacks at runtime based on system-centric policies (such as an app that has read access to user contacts must not communicate to an app that has network access). Specifically, it uses a graph representation of the system to illustrate all possible communication channels among installed apps and system built-in components, and determines whether an action at runtime will incur an edge that would establish a policy violating channel in the graph. By runtime monitor and analysis of communication links across apps, XManDroid can also effectively detect the channels established through the Android system services and content providers, and even prevent the privilege escalation caused by collusion attacks via covert channels. However, it requires the policies for decision making to be clearly and widely defined to prevent all sorts of attacks that exploit ICC channels.

Above solutions indeed offer more flexibility for users to confine the resource access and communication among apps, but also increase the complexity of the default permission-based model. Most of them eventually rely on the proper policies to be effective in practice. It is non-trivial to define these policies. Thus, instead of fighting with the permission-based model, another direction for resource protection starts from the point of view of the resources to be protected.

2.2.2 Reinforce Data Protection through Isolation-based Approaches

While a system supports versatile functions and integrates rich resources, the isolation technique plays an important role in sensitive resource protection. This technique is mature in the conventional desktop environment. Below we will first introduce the brief idea of traditional isolation-related techniques. We also discuss the needs and variations for such techniques to be migrated into the mobile platform.

2.2.2.1 Sandboxing

Sandboxing techniques target at building a secure and tightly-restricted environment for execution of untrusted applications. Any access across the environment boundary is either denied or regulated. Here we introduce a few closely related system call interposition-based sandboxing approaches in the traditional desktop environment.

Janus [49], proposed by Goldberg et al., is one framework for sandboxing the execution environment for helper applications. They created several policy modules to check each matched system call whether it should be allowed or denied. The available policy modules are selected by the configuration file for specific application. For example, path module checks whether file access is allowed according to the file location, and tcpconnect module restricts the allowed TCP connections. Each policy module is responsible for specific functionality and could contain several related system calls. One system call could occur in more than one policy module. According to their policies, the privileges of untrusted helper applications are restricted under one file folder, and any priviledge elevation, like setuid operation, is simply discarded.

Systrace [81], proposed by Provos, is an extensible system call interposition tool. It provides flexible interfaces to dynamically generate security policies automatically and interactively. Additionally, privilege elevation is also integrated into policies for single system call. It intercepts at system call entry and exit, and enforces security policies during application execution. Any system call path which deviates the security policies will be denied and recorded. The security policies are generated automatically in the training phase, which could include most possible benign behaviors of an application. Policies are listed according to different types of system calls by enumeration, so it is easy to append new policies dynamically. Users could also interact and refine the policies or terminate the application when any uncovered system call path is executed before security policies are finalized. To avoid the complexity of the policy language, they define their own policy statement rules to satisfy the application needs by only permitting necessary system calls. Each system call has a list of policy statements. Unlike other intrusion

systems which only audit a sequence of system call names, their policy statements include more system call semantics. Once matched, policy statements relevant to the matching system call will be checked in order. The first matched policy statement either denies or permits this system call, or asks users to determine explicitly.

Etrace [65], proposed by Liang et al., is another extensible system call interposition framework. It intercepts each system call made by the monitored process and all its children. One advantage is that it hides the low-level details and extracts general interposition interfaces. Therefore, it is portable across platforms as long as developers rewrite the architecture-dependent implementation. It also provides interfaces for developers to introduce their own extensions into the Etrace framework so that they can handle each system call event and enforce their security policies conveniently.

Towards the mobile platform, essentially the concept of sandboxing has already been integrated into the design of the Android platform. The Android platform, in which all the apps are executed in their own separate contexts, has made lots of efforts on process isolation. Each process can only access its own files, except those files shared with others explictly by users. Each app has to request strict permissions to ultilize system resources, such as acessing files and sending network packets. However, it is not sufficient to fully isolate untrusted apps from tampering or disclosing users' private information. We can not fully rely on customers' choices to grant permissions to third-party apps that may be greedy to require excessive and unnecessary permissions. Previous desktop-based sandboxing solutions [81, 49, 66] propose system call level interposition to provide fine-grained constraints on system resources. However, little research has yet targeted on applying similar techniques into the Android platform to enforce tightly-controlled security policies for untrusted apps. Providing a tightly-controlled sandbox environment is still an interesting explorable area in the mobile platform.

2.2.2.2 Virtualization

Virtualization is an important technique to achieve isolation. Through systematic virtualization, we can duplicate several virtual and separate environments, which are transparent to the apps running inside them. In this way, we do not need to change the default permission-based model in Android, which gains a good compatibility with the existing framework. Below we introduce a few representative solutions in this direction.

L4Android [64] runs the Android system completely in an independent virtual machine on top of a micro-kernel. Based on the L4Android micro-kernel, multiple Android systems can exist simultaneously and independently, each one inside a separate virtual machine. In their framework, the micro-kernel acts as the secure foundation and is augmented with a user-mode runtime environment that implements basic major operating system infrastructure and offers generic and known interfaces to applications. The framework supports a unified corporate and private mobile phone by multiplexing shared devices by both Android instances, such as smartcard, graphics hardware and input devices. However, L4Android's heavy overhead makes it not ready for resource-limited mobile devices.

Cells [16] proposes a virtual phone (VP) environment through light-weight operating system virtualization in one system instance. Cells introduces device namespace through both kernel-level and user-level device virtualization, and controls the device usage through wrapping device drivers, such as framebuffer/GPU, various sensors and even telephony subsystem (supporting multiple phone numbers by paring Cells with a VoIP service). One VP represents an isolated environment with several available virtual hardware devices. A physical phone can have several VPs. Users can easily switch among VPs by selecting one as the foreground VP while keeping the rest running in the background. The system can even force the switch as a result of an event, such as an incoming call or text message.

AirBag [100], a light-weight OS-level virtualization, isolates and prevents malware from infecting the system or stealthily leaking private information. It builds a restricted

execution environment for untrusted apps and also virtualizes various device drivers, such as file system, framebuffer/GPU and input devices. The goal is to mediate the access to various system resources or phone functionalities by malicious apps through instrumenting related API calls to the native Android runtime and multiplexing context-aware devices. It provides a separate namespace and filesystem to restrict and isolate the capabilities of processes running inside by leveraging Linux kernel containment features. To boot up AirBag, it launches the same subset of service processes or daemons (e.g., void, binder) as when the Android system is loaded, thus owning a separate Android framework in its own runtime.

TrustDriod [25] proposes an even light-weight virtualization architecture by intercepting the communication and data access in both middleware layer and kernel layer, without duplicating the Android software stack. It groups apps into two different domains, trusted corporate domain and untrusted domain. It designs a coloring mechanism to separate and distinguish apps and user data that belong to different domains. The communication among domains is restricted to prevent the data leakage. Additionally, if any data is written into the system services or content providers by one app, only apps in the same domain with it can read the data, otherwise can get a pseduo or null value alternatively. TrustDroid also applies this rule on the system-wide shared file system so that a system-wide readable file can be only read by another app of the same color as the writer. This solution is rather light-weight, but does not provide imperceptible transparent virtualization on certain resources, such as one domain still perceives the existence of files belonging to the other.

Above solutions propose virtualization-based solutions in different levels. They build a separate virtual environment for executing apps and accessing resources. However, for most of them, the cost to create such an environment (including the environment initialization, extra memory and storage) is a concern, especially on the resource-restricted mobile platform. TrustDroid is rather light-weight, but fails to gain the virtualization on certain resources (e.g., file system) due to their access-control-based design. Towards the

goal of resource protection, we can further reduce the virtualization cost by only focusing on the sensitive resources, instead of designing a complete full-fledged virtualization framework.

2.2.2.3 Partition

Partition is an alternative concept to separate sensitive resources into a different partition where we can apply a tight-control mechanism on the resource access. Based on this concept, a few solutions have been proposed and applied in various contexts, such as separation between ads and the main process, separation through a trusted execution environment.

AdSplit [92] extends Android to allow an app and its advertising libraries to run in separate processes, eliminating the need for apps to request permissions on behalf of their advertising libraries and also preventing forging the user interaction and stealing money from advertiser by malicious apps. In their framework, ads libraries and their hosting app have different user-ids, which are recognized as two separate apps by the Android system. All advertising libraries are loaded alongside the hosting app but with different user-ids. Therefore, the separation of the runtime environment (including the permission, process, private data) is provided by default by the existing Android sandbox mechanism. It designs the mechanism for screen sharing and authenticated user input by leveraging the QUIRE framework.

AdDroid[76] separates the advertising libraries and the hosting apps by migrating the advertising support into the Android framework. Therefore, with the extended Android API, apps only need to specify in the configuration to indicate from which advertising network to fetch ads. Different with AdSplit that starts lots of new processes for loading ads libraries, AdDroid adds a new dedicated system service in the Android system which is only responsible for receiving advertising requests from apps via the extended API and returning ads. In this way, apps only need to issue an ICC request to this dedicated service for advertising. They propose additional permissions `ADVERTISING` and

`LOCATION_ADVERTISING` to restrict the ICC with the AdDroid service, which are the only permissions requested for advertising in apps, thus mitigating the overprivilege problem caused by advertising libraries.

Additional to the context of partition between advertising libraries and hosting apps, partition is also applied to provide new architectures for separate trusted execution environment, especially on the hardware level. The recent ARM-based architecture supports a new security extension, named as ARM TrustZone, which partitions the hardware resources on the platform into trusted secure world and untrusted normal world, and enables CPU to run in either of the mode and switch between the worlds. This is a promising security framework that provides strong isolation guarantees for the trusted components with high security requirements, such as online payment module and credential management. Our root of trust only relies on a small runtime environment which is completely separate with any commodity OS running inside untrusted domain, thus gaining a small trusted computing base (TCB) and high resistance to threats from even a compromised Android OS. Limited research has been done to take advantage of this architecture for resource protection on the Android platform. Below we introduce a few designs by recent work based on this architecture to propose general frameworks for the ARM-based system.

TLR [85] (Trusted Language Runtime) is such a design on the ARM TrustZone architecture but for .NET mobile apps. It protects confidentiality and integrity of .NET mobile apps from OS security breaches. It enables separating an app's security-sensitive logic from the rest of the app, and isolates it from the OS and other apps. The secure world provides a runtime environment for a customized .NET micro framework supporting high-level languages like C# for ease of programming. It only supports computation with no access to peripherals to keep a small TCB. In their programming model, developers must instantiate an instance of the secure runtime environment (*Trustbox*) for security-sensitive part (*Trustlet*) of an app. The interaction for the two worlds is wrapped as the way of a simple procedure call for apps through pre-defined library interfaces, which hides the complicated procedure for connecting the two separate parts of an app, such as world

switching, interrupt handling and call routing.

ObC [62] (On-board Credentials) designs an architecture for the credential management on the ARM TrustZone architecture, currently only for Symbian OS on Nokia phone, and provisions credential secrets that are only accessible to specific pre-authorized programs inside the secure environment. Many current systems for user authentication either require users to memorize passwords or rely on dedicated hardware tokens, suffering from bad usability. Towards this problem, ObC designs a credential management system using the ARM TrustZone secure hardware that balances flexibility and high levels of protection. Users do not need to struggle with the passwords or tokens with the design of ObC in which all the credentials are protected by the underlying secure hardware. Due to the resource limitation in the secure world (e.g., limited RAM), they deploy a simple bytecode interpreter for Lua as the secure runtime environment. Only credential programs are allowed to execute inside ObC interpreter and thus perform sealing/unsealing actions. It also provides the provisioning architecture based on PKI (Public Key Infrastructure) mechanism, which allows the openness of provisioning new credential secrets and programs to users' devices without any third-party approval while still preventing malicious credential programs from stealing other credentials on the same device.

These hardware-assisted partition-based solutions provide strong isolation guarantee for the secure environment. The secure environment usually only contains limited resources and inevitably has limitation of supported functionality. We can only deploy a small portion of security-sensitive code into the secure environment. For example, in above solutions, they choose to customize and shrink the secure runtime environment through either limiting the access to peripherals or using a slimmed down version of simple interpreter. Existing research related with trusted execution environment on the Android platform is rather limited. This is an explorable and promising domain that can be further applied as a strong support for resource protection on the Android platform.

2.2.3 Common Android Malware Detection

Most of existing mobile anti-virus softwares rely on known malware samples for signature extraction. Traditional signature-based scanning is efficient but also has limited effectiveness. Such signature scanning is easily defeated with encryption and polymorphism. Behavior-based analysis is the most common approach to spot zero-day Android malware. Considering the energy constraints on the mobile platform, traditional heavy host-based detection is limited. Thus, an emerging proposal to sidestep the energy constraints uses offloaded architectures. Furthermore, on the Android platform, it is common that known trojans are repackaged into popular legitimate apps and spread in the market. Thus, some emerging techniques have been proposed towards detecting and distinguishing the repackaged malware. Next, we will discuss the representative work towards malware detection.

Paranoid [47], proposed by Protokalidis et al., provides a creative offloaded architecture in which all the events in the host are kept synchronized with a well-provisioned server in the cloud. On the server, it runs an emulator to replay the trace file received from the host Android device. Therefore, all the further security checks are performed on the powerful server. It is even compatible with heavy desktop-based security model, like dynamic program analysis, without any concern about the restriction on energy consumption. To reduce redundant and unnecessary network transmission, they only report undetermined system call operations to the cloud and also give an optimization for lazy transmission when huge amount of data are generated, like files and photos. However, in general, these offloaded architectures rely on a stable network channel with remote servers and also incur extra power expenditure due to data upload, which are hardly applied widely in the current smartphone market.

RiskRanker [52] is a kind of static behavior-based analysis to scalably and accurately sift through a large number of apps from the existing Android market to uncover zero-day malware. It analyzes whether a particular app exhibits dangerous behavior (e.g., launching a root exploit or sending SMS messages in the background) and produces a

prioritized list of reduced apps that merit further investigation. Specifically, it proposes a two-order risk analysis to assess the security risks of Android apps and classify them into high- or medium-risk based on the malicious behavior patterns. The first-order sift is designed to handle non-obfuscated apps in a straightforward manner. For example, it treats apps as high-risk if they match the signature of known platform-level exploits (e.g., Asroot, Exploid, GingerBreak), and treats apps as medium-risk if they charge users' money surreptitiously or update undeniably private information to a remote server through both control- and data-flow analysis techniques. To deal with obfuscated code, they further develop second-order analysis to collect and correlate various signs or patterns of behavior to identify encrypted native code execution and unsafe Dalvik code loading. By applying their technique on 118,318 apps with less than four days, they uncover 718 malware samples and 322 of them are zero-day malware. They also measure the false negative, which is mainly due to the unmatched behavior pattern or the difficulty of distinguishing malicious and legitimate apps from the common behavior such as information collection.

Crowdroid [26], proposed by Burguera et al., is a dynamic behavior-based detection approach, providing a framework to distinguish between apps that, having the same name and version, behave differently. Capturing and analyzing the system calls at the kernel level produces accurate information about the apps' behavior. System call sequence monitoring has been widely used in intrusion detection system. Therefore, Crowdroid collects related information (like opened and accessed files, execution time stamps and the count of invocation for each system call number) about all the system calls based on the *Strace* tool, and sends them to a dedicated centralized server for information processing. By calculating the vector distance of the collected system call feature vectors with benign traces, Crowdroid effectively distinguishes the malicious traces generated by repackaged malware samples, and also summarizes a few favorite system calls by trojanized apps.

In general, signature scanning and behavior analysis are the two main approaches for malware detection. Most of anti-virus softwares only apply signature scanning to achieve efficiency. However, they are susceptible to common evasion techniques, such as obfusca-

tion, encryption and repackaging. DroidChameleon [83] evaluates the most popular anti-virus softwares against transformation attacks and reveals that all of them are vulnerable to common transformations. A considerable amount of them only check checksums and configuration files without any code-level analysis. Behavior-based analysis essentially aims to distinguish malware from benign apps according to different behavior patterns. However, it falls into false negative for common behavior, such as information collection which also frequently occurs in benign cases.

2.2.4 Analyze How Applications Use Sensitive Data

To make thoughtful security-related decisions against threats to resources, we need more comprehensive understanding regarding the resource usage inside Android apps, in addition to signature-based information. Having the knowledge of how apps use the sensitive data, we can determine the potential risks of a given app on sensitive data, and thus conclude at which scenario or to what kinds of apps we should grant the resource access. Android markets can also use this knowledge as an important reference to check the potential risks of newly submitted apps and take effective measures to suspend suspicious apps from being widely spread at the first place. In this section, we elaborate data oriented analysis from different angles in present work.

2.2.4.1 Taint-based Data Flow Analysis

Taint is one important technique in the area of information flow analysis. It keeps tracking the information flow during data propagation. Below we list the representative work on both dynamic and static taint-based solutions.

TaintDroid [37], proposed by Enck et al., proposes an integrated tainting solution to track private information and monitor whether privileged applications are handling private data properly. This is similar with the conventional desktop-based tainting analysis, but a variant to satisfy strict requirement in smartphone hardware. It leverages four granularities of taint propagation: variable-level, method-level, message-level and file-level,

in terms of performance and accuracy. They mark private data with taint tags and propogate taint tags within the VM interpreter. They also instrument the Android framework libraries to initialize the taint marking and detect whether any suspicious function, like the network send API, is invoked to disclose tainted private data.

AppFence [56] extends the TaintDroid framework and prevents sensitive data leakage at runtime. It proposes data shadowing and exfiltration blocking techniques to protect resources. It blocks network APIs when detecting sensitive data being sent out through sockets, and alternatively shadows the content provider with an empty set and other primitive data (such as location and device ID) with certain fixed values. Specially, they evaluate the effectiveness of their approach on real-world samples. It shows that their countermeasure is a promising privacy control to reduce sensitive data exposure. 66% of apps in their testbed (50 in total) are compatible without side effects, while the rest have a direct conflict between the desired functionality and the privacy constraint.

FlowDroid [17] is a static taint analysis tool for Android apps, which aims to achieve high precision compared to existing other taint-based analysis. Therefore, their data flow analysis is designed to be context-, flow-, field- and object-sensitive for achieving high precision, and also takes the model of the Android-specific app lifecycle into account to properly handle callbacks invoked by the Android framework. FlowDroid leverages the callgraph and flow analysis by SOOT, an existing framework for optimizing Java bytecode, and then analyzes if there is a source-to-sink connection.

Generally, it is inevitable for taint-based solutions to suffer from either taint explosion or taint loss. Additionally, it aims to detect the presence of a flow from pre-defined sources to sinks, which only partially reflects the way of data usage. Users still have little knowledge about the apps' internal logic on utilizing certain sensitive data, which is also a helpful indicator, such as whether apps leak the raw sensitive data or only a little of them.

2.2.4.2 Symbolic-execution-based Analysis

Another line of work for privacy leakage detection is the symbolic-execution-based technique. It can be used to analyze under what conditions the data can be leaked.

SymDroid [57] designs a symbolic executor for Dalvik bytecode. SymDroid can be used to determine the conditions under which certain privileged calls (such as a privileged call that uses `READ_CONTACTS` permission) would be invoked by a chosen activity. They first translate Dalvik (supporting more than 200 instructions) into a simple version, i.e., μ-Dalvik, which has only 16 instructions. Dalvik bytecode is designed to reduce the overall size of apps and runtime performance in a resource-constrained environment, while μ-Dalvik provides as clean and simple semantics as possible for symbolic execution that is a possibly expensive off-device analysis. They model some key portions of the Android platform (common libraries and lifecycle control code) and design a small number of standard and quite straightforward operational semantics rules for the symbolic executor. Due to lack of a complete system model, it only passes 26 of the test cases in their testbed (93 in total).

AppIntent [106] proposes a more efficient event-space constraint guided symbolic execution approach, considering the special event-driven paradigm in Android. They claim that whether a data transmission indicates privacy leakage should eventually depend on users' intention. Therefore, they first use static taint analysis to identify all possible transmission paths, and then use symbolic-execution-based solution to extract app inputs to trigger a given sensitive data transmission path. The path condition helps analysts to determine whether a transmission of sensitive data is user-intended or not and thus identify the privacy leakage more comprehensively. Specially, to deal with the path explosion problem in symbolic-execution-based solutions, they propose an event-space constraint guided symbolic execution mechanism by leveraging the Android event-driven system.

SpanDex [84] borrows techniques from symbolic execution to precisely quantify the amount of information (e.g., users' passwords) that a process' control flow reveals. It enhances TaintDroid to handle the implicit flows and applies pre-defined heuristics along

the data flow to calculate the possibility of revealing the sensitive password from the result of a mathematical or branch operation. It ensures that the amount of secret information revealed through a process' control flow does not exceed a safe threshold. However, the solution is rather limited to handle real-world complex scenarios, such as cryptographic libraries, bit-wise and array-indexing operations, and even multiplication and division.

Symbolic execution is usually time-consuming for a full-fledged analysis to explore all the feasible paths in one app. It needs an interpreter to maintain the program state. If we target only on how apps use data, this technique is relatively heavy and impractical to handle large-scale real-world Android apps.

2.2.4.3 Program-slicing-based Analysis

Program slicing is a common technique for data dependency analysis, which is relative light-weight comparing to above solutions. Some existing work applies this technique on data dependency in bytecode-level.

SAAF [55] is a slicing-based bytecode-level static analysis framework for Android apps. They consider it to be suspicious if an interesting method uses a constant as its parameter. For example, SMS sending API sends a message to a fixed phone number. It aims to create program slices in order to perform data-flow analysis to backtrack parameters used by a given method. SAAF unpacks the Android apps and disassembles all classes into *Smali* format, an "assembly-like format" for the Android bytecode. SAAF then parses the Smali files and creates appropriate representations for all the contents, such as basic blocks of the methods, fields and all opcodes. SAAF performs their customized program slicing logic for backtracking analysis. SAAF backtracks the parameters of pre-defined interesting APIs and checks whether the parameters are related with constants.

Program slicing is an alternative direction for data usage analysis. It reveals the data dependency in a program slice, which can represent what kinds of operations are performed on the sensitive data. Due to its light-weight nature, it is suitable for large-scale analysis. It is worth thinking about how to balance the analysis granularity and the valu-

able information extracted from the apps regarding how apps use the sensitive data.

2.3 Summary

In this section, we briefly introduce the Android infrastructure and the existing solutions along the direction of resource protection on the Android platform. We discuss the related work on enhancement and analysis frameworks from four angles, which are 1) enhancing the existing permission-based model in Android; 2) achieving data protection through isolation-based approaches; 3) detecting malware threats to sensitive resources; 4) determining the nature of an app on how it uses sensitive data. Especially, we discuss the present work and the promising direction in virtualization-based and partition-based approaches for data protection. This leads us to the thought of a better trade-off between data protection and the sacrificed usability and overhead. The existing work on data usage analysis also helps us to think of other important complementary metrics, such as the impact of a set of operations on sensitive data indicating whether they leak more or less of user privacy, which are also key factors for users to better understand the apps' data usage.

Chapter 3

A Light-weight Software Environment for Confining Android Malware

3.1 Introduction

Mobile devices (e.g., smartphones and tablets) have become increasingly general-purpose and versatile. A mobile device integrates the functionality of several special-purpose devices, such as a mobile phone, a GPS, a game console, an e-book reader and an Internet tablet. While such functional integration significantly improves user experience, it also increases system complexity. In such highly-integrated devices, as the physical barrier between many types of resources disappears, mobile applications (apps) may exploit vulnerability and misconfiguration to access important resources that they should not have accessed. Recent research [37, 56, 113] has revealed that a large portion of apps expose phone state or location information. Malicious apps can be designed to covertly extract private information, such as user contacts, or stealthily capture audio/video through microphone and camera services [88, 102]. Even worse, trojans, such as *Geinimi* and *DroidDream* [7], are repackaged into popular games and get widespread, which almost gain full control over infected Android devices. Although the official Android market actively detects malicious apps, malware has been spotted from time to time in widely used apps.

Malware is a more severe problem in third-party markets, where resources for detecting malware are limited.

To regulate an app's access to the resources on mobile devices, the Android system uses a permission-based security mechanism. Apps need to explicitly request permissions for resources, and the requests will be decided by users during the installation time. Under this mechanism, users only have two options to refuse installation of a suspicious app or to completely grant all the permissions requested by the app. Since users are often not knowledgeable enough to understand the Android permissions, malicious apps can misuse resources unintended by users.

To improve the default permission-based model in Android, recent solutions [74, 20] have extended it to support more flexible permission control on apps and allow users to selectively grant permissions or re-adjust permissions for an app after installation. However, it is tedious for users to selectively grant permissions and also impractical for users to make all security-related decisions. Even with properly configured permissions, malicious apps can misuse resources through vulnerabilities in the Android inter-component communication (ICC). Although recent work [51, 35, 23] proposes solutions to detect and prevent ICC vulnerability, they are not general for confining resource-abusing apps without ICC channels involved. A stronger security guarantee is provided by virtualization-based solutions — Cells [16], L4Android [64], and AirBag [100] propose OS-level virtualization and isolated execution environment by virtualizing hardware devices. However, they require heavy environment initialization and extra storage to achieve strong isolation. In this work, we propose a light-weight virtualization solution which balances between the security and usability for confining resource-abusing Android apps.

To effectively mitigate the resource-abusing problem and manage the complexity of protecting Android resources, we propose an approach based on the idea of resource virtualization. It creates a *resource virtualization layer (RVL)*, which sits between the mobile apps and physical resources (e.g., the raw external storage and user contacts). Apps can only obtain a virtual view of the resources, instead of directly accessing the physi-

cal ones. Our approach confines apps within an isolated software environment, a *virtual device*, in which apps get a "private" copy of system resources. Our solution supports multiple isolated environments and allows users to customize each of them. Therefore, a general-purpose device can play roles of a virtual e-book reader and a virtual Internet tablet simultaneously without affecting each other.

Our approach achieves a balance between usability and security guarantee without incurring heavy overhead. Our goal is to provide an isolated clean environment by default, instead of restricting any functionality by permission removal. We further allow advanced users to customize the virtual view of the resources inside each environment. Comparing to figuring out the meaning of the permissions requested by various apps, it is easier for users to understand what resources should be provided for the expected functionality. For example, when a user wants to download a media player app, he can create a virtual media player device with only storage and Internet support. Then he can simply install several media player apps into the virtual media player device without struggling with the permissions for each app and worrying about the potential resource-abusing problem. Our solution can also be flexibly applied into centralized enterprise management in which the administrator pre-defines a set of apps along with the environment configurations. Comparing with other OS-level virtualization solutions, RVL does not duplicate any physical resource, such as the external storage and user contacts. Additionally, creating a new virtual device environment only requires small changes on configuration files, without heavy environment initialization or booting process. Therefore, RVL incurs negligible storage and memory overhead. It only shows 0.25%$\sim$3.87% runtime performance overhead in our performance evaluation on popular Android benchmarks.

We prototype RVL in the Android system and evaluate it using real-world Android apps. Our experiments show that RVL can effectively virtualize Android resources and successfully isolate Android apps while preserving their functionality.

In summary, our contributions are as follows:

- We design a light-weight resource virtualization layer, RVL, in the Android system

to mitigate the resource-abusing problem. It balances security and usability.

- We evaluate RVL using real-world Android apps (including both benign and malware samples). RVL can effectively prevent malicious behaviors without compromising the usability of these apps.

3.2 Approach Overview

In this section, we introduce the resource protection mechanism in the Android system and the resource-abusing attacks. We then give an overview of our solution.

3.2.1 Android Resource Protection

The Android system enforces a permission-based security model. An app needs to declare a list of permissions that must be granted for accessing resources. For example, in order to gain the capability of sending SMS, the app must specify the `SEND_SMS` label in the *AndroidManifest.xml* configuration file. During app installation, the Android system prompts this permission list, and users must decide whether to grant all the permissions to install this app, or refuse the permissions to deny the installation. The Android system also checks permissions at runtime at a number of places, such as starting an activity and binding to a service.

However, the current permission-based security model is insufficient in the following aspects.

1) It relies on users to make proper security decisions. Users decide whether to install an app by examining its permission list. It is extremely difficult for end users to understand the precise meaning and security implication for each permission. Even if some users do have good understanding on these permissions, it is still hard to decide whether an app can be harmful, given only the list of requested permissions.

2) Users cannot selectively grant a subset of permissions requested by one app during installation. After an app gets installed, users cannot later revoke a granted permission. It

means that once an app gets installed, users have no control over how it utilizes granted permissions and whether it performs malicious activities. For example, if one app requires both location and network permissions, it can easily leak location information to a third-party. Even though the Android 4.3/4.4.1 releases have a hidden feature "Apps Ops" which allows users to dynamically revoke permissions for an app, this feature is not complete yet and may break certain apps after permission removal. Thus Google completely disables this hidden feature in the Android 4.4.2 release.

3) Permission labels are coarse grained and not expressive enough for certain scenarios. For example, most Android devices support SD card as the external storage. In order to gain a full access to the SD card, an app must hold `WRITE_EXTERNAL_STORAGE` permission. However, the SD card resource is shared among all the apps with that permission on the device. Consequently, if a malicious app obtains this permission, it can corrupt other apps' data or code in the SD card. Users cannot grant access to a portion of the SD card file system to an app and deny its access to the rest of the SD card.

4) Apps can reuse components provided by other apps, thus resulting in permission escalation problem. Through the ICC between apps, a malicious app can escalate its privilege or inject malicious input to corrupt other benign apps [51, 24, 35, 75, 23]. For example, a privileged app having the permission of sending SMS can send SMS for other unprivileged apps, so that malicious apps can request this privileged app to send SMS through the ICC channel without requesting SMS-related permissions by themselves. Thus, even though an app seems benign by itself, it may bring hidden vulnerable channels and reduce security of the whole system after being installed into a device.

Resource-abusing Application Example. According to [110, 111], the majority of real-world malware samples (about 86%) are from repackaging existing legitimate apps with malicious payloads. For example, *Geinimi* trojan [7] is repackaged into popular game apps, e.g., *Monkey Jump 2*, to attract users. Once infecting a device, the trojan steals sensitive data, such as user contacts, SMS messages and location data. Usually, the repackaged games request more dangerous permissions for those sensitive resources during installa-

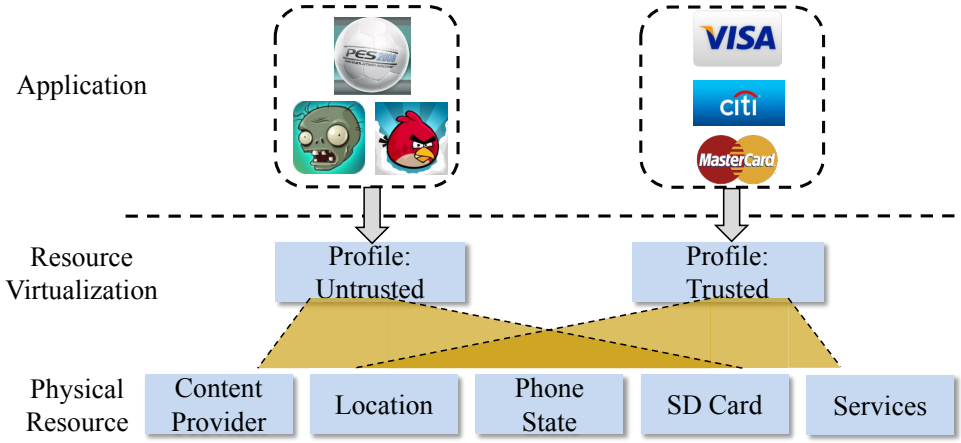


Figure 3.1: Android Resource Virtualization

tion than normal games. One intuitive detection method is to examine the permissions requested by the apps. However, it is difficult to decide whether an app is malicious simply based on the requested dangerous permissions. A malicious app only behaves maliciously at some point during its execution. Therefore, we need a systematic protection mechanism to mitigate the threats to an Android device from installing a malicious app.

3.2.2 RVL Overview

The main idea of our approach is to virtualize resources on the Android platform, such as the user contacts database and the SD card file system, and to provide a separate *resource virtualization layer*. Installed apps will access system resources through this virtualization layer, instead of directly accessing the physical resources. Based on this virtualization layer, users can define *profiles* to isolate apps. Apps in each profile share the same view of resources, and apps in different profiles can neither share resources nor communicate.

Threat Model. Our goal is to mitigate resource-abusing problems with minimal modifications to the Android system. The TCB (Trusted Computing Base) for our solution includes the OS kernel and the Android framework, which is equivalent to the TCB of the original Android security model. This implies that a malicious app with root exploits is out of the scope of this approach.

In the example illustrated in Figure 3.1, we show two profiles, `Untrusted` profile

and `Trusted` profile. Note that we just use this example to demonstrate one applicable scenario of the profiles, not to discuss how to group apps based on app trust. Our solution can also be applied in other scenarios, such as `Work` and `Personal` profiles for BYOD (bring-your-own-device) paradigm. By default, each profile is an isolated virtual device environment with its own virtual private copy of sensitive Android resources, such as the user contacts and external storage. For example, to the *Angry Bird* app in the `Untrusted` profile, it seems as if it runs in a physical device with sensitive Android resources available, except for perceiving the existence of apps in the other profile. Apps in one profile cannot access resources of the other. Modifications made by an app only take effect within its own profile.

RVL provides a separate virtual view of physical resources for different profiles based on profile configurations. The default configuration guarantees that each profile has its own private virtual copy of resources. Users can also flexibly customize the default configuration on resource access, so that they can manually group the installed apps based on app trust or specific resource control, such as whether to allow the network/contact access or not. The resources of a profile can be dynamically changed, such as on-demand privileges. Users can create multiple customized virtual devices running simultaneously above the physical one. Through RVL, apps in different profiles will have different views of phone resources.

RVL is transparent to the confined apps. For compatibility of existing apps, RVL does not conflict with the existing Android permission-based security mechanism. It just wraps physical phone resources. Apps will not see any change from the interfaces they depend on. For example, in Figure 3.1, if the *Angry Bird* app in the profile `Untrusted` wants to access the user contacts database, it still needs to claim `READ_CONTACTS` permission label in its *AndroidManifest.xml* explicitly. Even if the permission is granted by the user, the app can only access the virtual contacts database in the `Untrusted` profile.

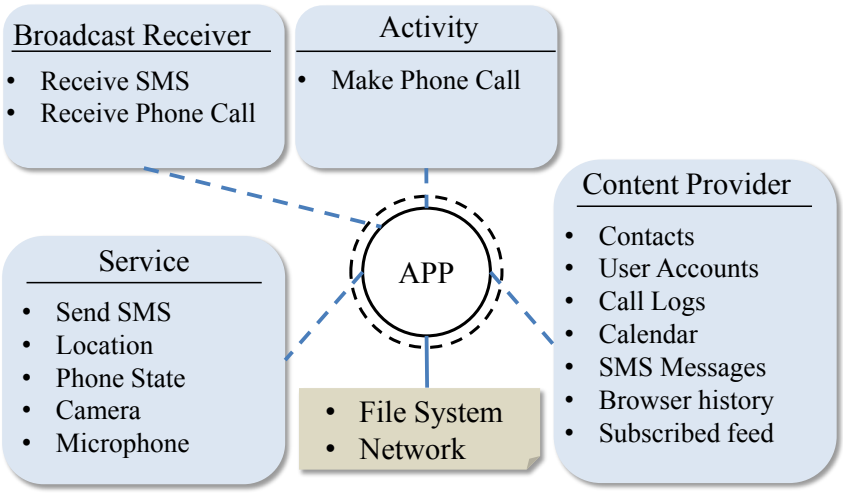


Figure 3.2: Android Resources

3.3 Resource Virtualization in Android

In this section, we describe our resource virtualization based approach in detail. We first summarize various types of sensitive resources in Android and then describe the technique to virtualize each type of them.

3.3.1 Resources in Android

The Android system is built up on the Linux kernel. Android apps are developed in Java language and compiled into `dex` code, a variant of standard Java bytecode. Each app runs in a separate custom Java runtime environment, the Dalvik virtual machine. To simplify app development, Android adds a middleware layer, which wraps local C libraries and provides well-defined Java interfaces for upper layer apps. Android also supports restricted JNI interfaces, which allow apps to directly import C libraries, such as those accessing file system and network. Resource access is controlled either by the Linux security mechanism — UNIX-style file access control, or by the Android-specific security mechanism — app permissions.

3.3.1.1 Linux System Resources

File system is one type of important resource protected by the Linux kernel. Usually, each app is assigned with a unique Linux user ID during installation. Linux associates each file with the app's user ID and group ID, and protects files using standard Linux file system permissions.

In Android, the device storage has two parts: internal storage and external storage. For the internal storage, most files belong to either the user `system` or the user `root`. These files can only be accessed by system services with high privilege. By default, Android assigns one separate private folder on the internal storage for each app. Usually, an app stores its own data, such as app-specific credentials and resources, in this private folder. Files created by the app will be assigned with the same user ID as the app. These files are not accessible by other apps with different user IDs.

Android devices also support the external storage as an extension to the internal storage. It can be a mounted SD card or non-removable storage media. The external storage is shared among all the apps. Typically, it is mounted and accessible through the `/mnt/sdcard` folder. The SD card file system is associated with the group ID `sdcard_rw`, where users inside group `sdcard_rw` have full read and write permissions. Android provides Java interfaces for accessing the SD card, and even supports JNI interfaces for apps to access the SD card directly using C libraries. If an app is granted the permission `WRITE_EXTERNAL_STORAGE`, the system will add the app into the `sdcard_rw` group, so that this app's SD card access will be allowed by the Linux kernel.

In addition, system drivers can create device files, so that communication with system drivers can be achieved in the same way as accessing an ordinary file. For example, *rild* is the system driver responsible for all the communication from the Android telephony services, such as SMS dispatch and phone call. The Android system creates a device file and only assigns the write permission to apps in the `radio` group. Android apps without the telephony privilege will not be added into the `radio` group so that the Linux kernel

can prevent them from accessing the telephony device file.

The network access is also confined by the Linux kernel in a similar way, that is, checking whether the running process is inside the `inet` group or not. The Android system adds apps with the `INTERNET` permission into the `inet` group and the Linux kernel ensures that only processes in the `inet` group can have the network access.

3.3.1.2 Android-specific Resources

Android apps can be decomposed into components. Four primary components are defined in the Android system: *Activity*, *Service*, *Broadcast Receiver*, and *Content Provider*. The Android system contains several built-in components, which are built into a device with the system image. These components take responsibility of managing different important resources, such as user contacts, calendar and SMS. For example, user contacts are maintained in a built-in content provider *ContactsProvider*, while an app needs to bind the built-in *ISMS* service to send SMS. Android provides well-defined interfaces for these components. For example, Android defines *SMSManager* for upper layer apps to easily send SMS without handling internal service binding. For most resources, an app has no privilege for direct access. The corresponding built-in component is the only interface for accessing them. An app does not need to understand the internals of each built-in component. It just needs to make a request to the corresponding component through predefined APIs. The component will check whether the requesting app has permissions for the requested resource. For example, *IccSmsInterfaceManagerProxy* registers one system service *ISMS*, which is in charge of sending SMS. If one app needs to send SMS, it must bind the *ISMS* service and ask the service to send SMS on its behalf through the *SMSManager* API provided by the Android framework. Similarly, if it needs to read user contacts, it must make a query to *ContactsProvider*, which is a built-in content provider component for managing contacts data. In Figure 3.2, we list those components that manage sensitive resources.

3.3.2 Light-weight Resource Virtualization

Integrating resources into one physical device introduces extra security risks, especially when users have little control on installed apps. Even though the permission-based security mechanism requests apps to explicitly apply permissions for resource access, it is still insufficient to effectively protect all local resources. We propose to virtualize physical resources and only provide a virtual view to upper layer apps. Apps running in such a virtual environment can only view virtual resources. We can virtualize multiple independent environments simultaneously above the physical device. For example, instead of all apps sharing the whole SD card and one single user contacts list in the standard Android system, we create multiple private copies of these resources and allocate one for each virtual environment, so that each virtual environment has its own external storage, user contacts, phone states, etc. Besides, apps running in one virtual environment are not able to view other environments' resources or communicate with any apps that belong to other environments.

Through resource virtualization, upper layer apps will obtain a virtual view of physical resources. We propose three virtualization mechanisms applied on different kinds of resources.

- For complicated resources, such as the *external storage* and the *content provider*, they are shared among many installed apps with read/write permissions. We propose to fully virtualize them to maintain consistency among installed apps.

- For simple resources managed by certain services, such as the location service managing the *location* information, the telephony service managing the *phone state* information and the *camera*, *microphone* services, they are read-only resources to apps. We propose to virtualize sensitive data by making simple changes.

- For other communication resources such as *send/receive SMS*, *make/receive phone call* and *network*, we will decide whether to support them in a profile.

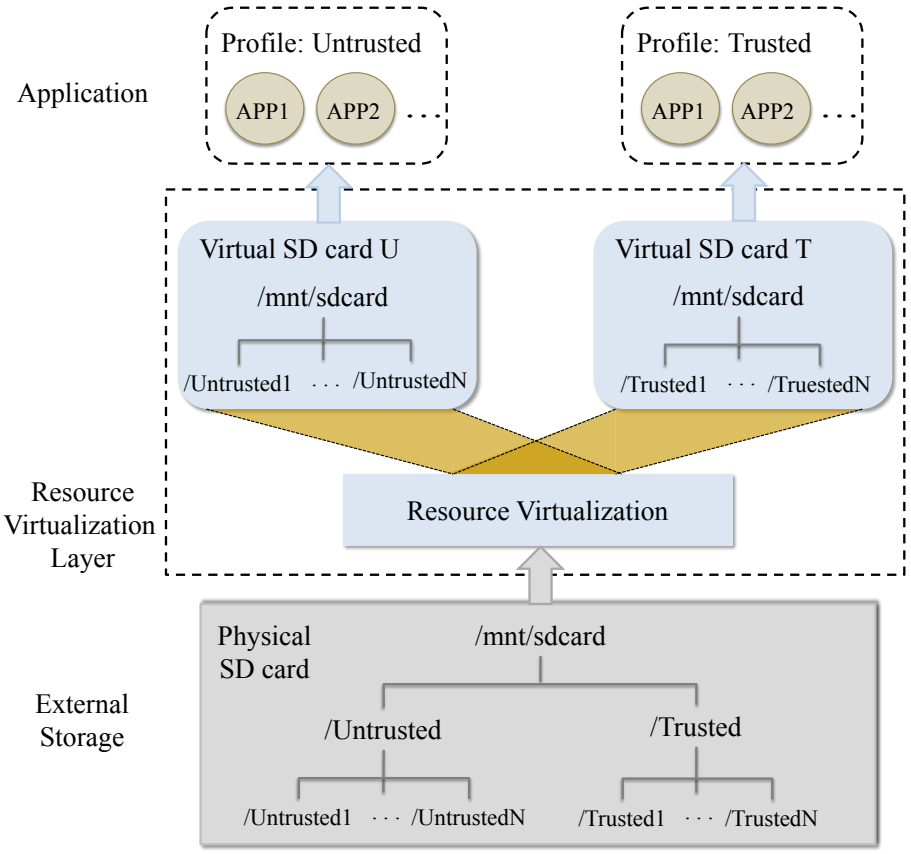


Figure 3.3: External Storage Virtualization

Virtualizing External Storage. The SD card is an external storage device currently supported by most Android devices. The SD card storage device is mounted in a public world-readable and world-writable folder, `/mnt/sdcard`. As long as an app is granted the `WRITE_EXTERNAL_STORAGE` permission, it can arbitrarily create new files or modify existing files created by other apps. Resources in the SD card are shared among all apps holding the permission, so if one app stores sensitive information into the SD card, other apps can directly corrupt or leak it. In addition, Android supports dynamically loading `dex` code during runtime from a given path using a standard API *DexClassLoader*. If an app simply uses an insecure path, like the SD card, to store `dex` code, it potentially gives other malicious apps a chance to replace the `dex` binary in the SD card with their own malicious `dex` code, so that malicious code can be executed on behalf of the victim app, as discussed by Poeplau et al. [78].

To mitigate the disadvantage from open access to the SD card, our approach provides

a virtual view of the SD card, so that apps are not given full control on the SD card. Figure 3.3 illustrates the virtualization of the SD card file system. RVL intercepts all the access from apps to the SD card file system and creates multiple separate and consistent virtual SD card file systems. Each virtual SD card file system aggregates one or more physical folders in the SD card. RVL redirects read/write operations of the virtual SD card to the corresponding physical folders and maintains a consistent view. For example, we use a physical folder `/mnt/sdcard/Untrusted` for apps in the profile `Untrusted`. All apps in the profile `Untrusted` will use `/mnt/sdcard/Untrusted` as the root directory of the external storage and cannot access beyond this folder's boundary. The processing is completely transparent to those apps. In this way, only apps within the same profile can share data in the SD card.

To ensure complete protection, we base our file system virtualization on the OS-level mechanism, i.e., system call interception to monitor the apps' processes. Through intercepting system calls for the file system access, we can constrain all access into a specific folder. In particular, our approach intercepts all file-system-related system calls of an app. The SD card in Android does not support link operations (such as symbolic links). It is usually mounted as the FAT file system. Since the Android version 4.2, it is more commonly mounted as a FUSE daemon but still without supporting link operations. Therefore, for files on the SD card, they have one-to-one mapping with the pathname. We can achieve a complete virtualization of the SD card by simply regulating the pathname of the file-system-related system calls. When a system call is invoked in the app, the system call is intercepted and forwarded to a file system virtualization engine. For a file-system-related system call, the file system virtualization engine checks whether the app belongs to any profile that is configured to constrain the SD card access. If so, it updates the pathname of the system call accordingly. After changing the pathname, it resumes the intercepted system call. For example, if an app in the `Untrusted` profile invokes an *open* system call to create a file with the pathname `/mnt/sdcard/temp.txt`, the *open* system call will be intercepted and

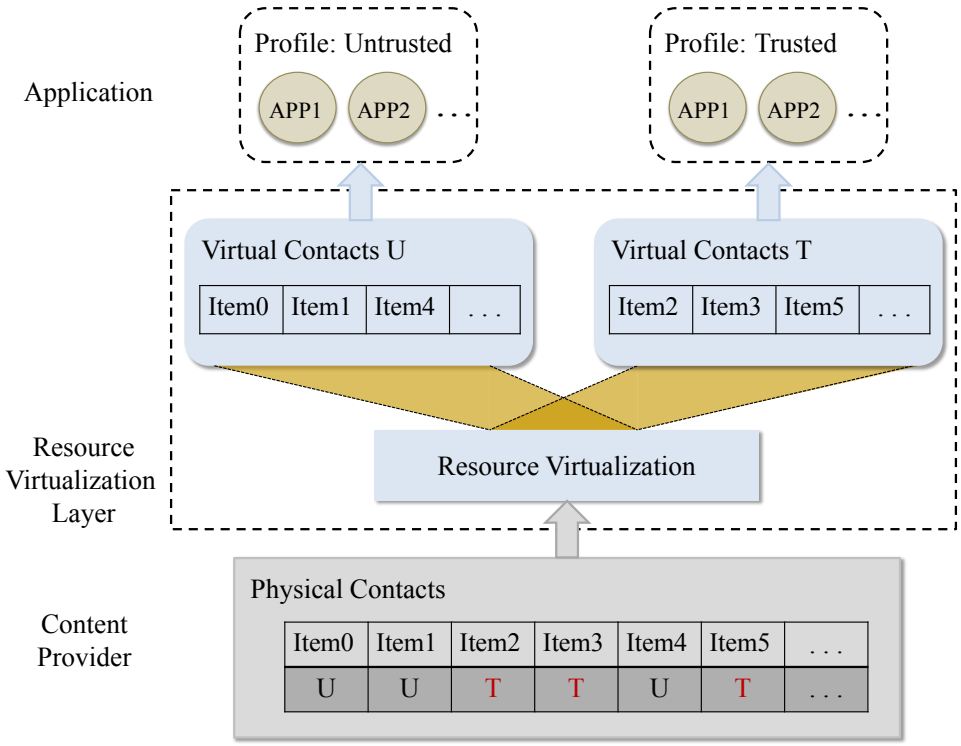


Figure 3.4: Content Provider Virtualization

redirected to the file system virtualization engine, which then automatically maps it to `/mnt/sdcard/Untrusted/temp.txt` with a pathname mapping. In the physical file system, the app creates a file `/mnt/sdcard/Untrusted/temp.txt`. If an app within another profile, such as `Trusted`, invokes *access* system call to read `/mnt/sdcard/temp.txt`, similarly the file system virtualization engine will map its pathname to `/mnt/sdcard/Trusted/temp.txt`. In this way, the SD card file system is virtualized and isolated.

Virtualizing Content Provider. A content provider is one type of Android components that manages data in the SQLite database format. The Android system provides several separate built-in content providers to manage user contacts, user accounts, call logs, calendar, SMS messages, browser history, subscribed feed, etc. Each content provider offers database-related operations, i.e., query/insert/update/delete, for apps to process the database. Each operation is also protected by the default permission-based mechanism in Android. For example, if one app wants to read the user contacts, it needs to request the permission `READ_CONTACTS` explicitly in its manifest file.

Our virtualization mechanism enhances default permission-based security of content providers by supporting tagging items in the database and regulating database access with respect to the tags. Therefore, two apps both with the `READ_CONTACTS` permission, but in different profiles, fetch different results through the same query on the same content provider. These tags are not visible to apps within any profile. If an app in a profile explicitly queries the tag information, it will get an Android *SQLiteException* indicating no such column.

Figure 3.4 illustrates the virtualization on content providers. Taking contacts information as an example, when an app in the `Untrusted` profile and an app in the `Trusted` profile access the contacts content provider through RVL, they will get different views that include different contact items. In the physical contacts content provider, items are marked with tags. The tag `U` indicates the corresponding item is visible to the `Untrusted` profile, while `T` for the `Trusted` profile. Assuming different profiles are querying all contact items, RVL revises the queries by adding one extra condition on those tags, and fetches different matching items for different profiles according to the profile configurations. In Figure 3.4, the `Untrusted` profile fetches all the items marked with the tag `U` while the `Trusted` profile fetches all the items marked with the tag `T`. Each profile is initially assigned an empty contact list, and when any app wants to insert a new item into the user contacts database, RVL automatically attaches the tag attribute, e.g., the item inserted by the `Untrusted` profile is marked with the tag `U` in the physical contact list, so that each profile can only view contact items created by apps in itself.

Virtualizing Simple Services. The Android system has many built-in system services which automatically launch during system boot time. Some of these services manage sensitive data, such as the location and phone state. Such services can be virtualized in a simple way, discussed below.

Location is the sensitive data managed by the *LocationManager*. To access it, apps need to request corresponding permissions and be explicitly granted by users during installation. Different from data in content providers, the location data is read-only to apps.

The GPS location information is generated by the GPS sensor, including latitude and longitude values. Apps registered as location receivers are notified through Java callbacks. To give separate virtual views, our approach assigns different values or reduces the accuracy of the raw GPS data, such as replacing partial of these data with '0' before being returned to apps. The inaccuracy degree depends on the profile configuration. For example, as the inaccuracy degree grows from low to high, the latitude value `37.422006` will be replaced as `37.422006`, ..., `37.0`, ..., until `0.0`. When a user searches for restaurants within around 500 meters using apps in the `Untrusted` profile while he does not want to leak his accurate location, he can configure in a way that apps in the `Untrusted` profile can only get the location information with certain degree of noise to be added.

A similar virtualizing mechanism is applied on the phone state information, which includes phone number, device ID (such as IMEI for GSM phones), etc. Some game clients use the device ID as user identification for the leaderboard on the server side. Sometimes, an app may take it for granted that the phone state information is available after being granted the `READ_PHONE_STATE` permission, so if we simply deny the access in a profile, the app may break. Even though users grant the phone state permission to a game app, they may just intend to agree that the app can use the device ID as user identification, while they may not expect the app to leak it out for other purposes. In this scenario, users can configure a separate virtual view on the phone state resource based on their requirements. For example, the `Untrusted` profile can only read inaccurate values or the hash values of the device ID and the phone number.

Camera and microphone are two services to collect media data. Considering practicality and performance, to virtualize the camera and microphone, we simply return a pre-selected image for camera and a pre-recorded audio for microphone.

Handling Communication Services. For sensitive resources, including making/receiving phone call, sending/receiving SMS and network, etc., we simply allow or deny them. In this way, we can mimic different virtual views of physical functionality support.

Apps need to go through Android-specific components in the middleware to access

these resources, such as starting an activity for making phone call and binding the corresponding service for sending SMS. Specially, for network access — Linux system resource, Android apps can directly access network without going through the Android framework layer. We need to rely on the OS-level mechanism to ensure the effectiveness through system call interception. We need simple confinement giving an allow/deny answer for network-related system calls.

3.3.3 Profile Configuration

$$\mathcal{P} := (\mathcal{ID}, \mathcal{AS}, \mathcal{RS})$$

$$\mathcal{AS} := \{\mathcal{A}_i | i \in 1 \ldots N_A\}$$

$$\mathcal{RS} := \{\mathcal{R}_i | i \in 1 \ldots N_R\}$$

Each profile $\mathcal{P}$ is represented as a tuple. $\mathcal{ID}$ represents the identification of one profile. $\mathcal{AS}$ represents the application set grouped into the profile $\mathcal{P}$, where N_A is the number of applications in the set $\mathcal{AS}$. One app can only belong to one profile at one time, but can be switched among profiles. $\mathcal{RS}$ indicates the virtual resource set inside $\mathcal{AS}$, where N_R is the number of different kinds of sensitive resources. $\mathcal{R}_i$ in $\mathcal{RS}$ represents the configuration policy for one type of resources at the index i.

Table 3.1 lists the available configuration options for different kinds of resources. *SDcard_View* has three options *none, partial, full*. Option *none* indicates that the corresponding profile is disallowed for any access to the SD card, while *full* indicates the full access to the SD card. Option *partial* means that the profile is given a virtual SD card view generated through RVL. Note that RVL is an additional protection layer in Android, and thus apps still have to satisfy the permission requirements for accessing sensitive resources even being grouped into a profile configured with *partial* or *full*. *ContentProvider_View* includes configurations for all different content providers, and users can configure separately for *Contacts_View, UserAccounts_View*, etc. Similarly, *ContentProvider_View* also has three options. Option *none* indicates denying all access to a certain content provider, while *full* indicates the profile can view the full content provider. Option *partial* indi-

cates that RVL generates a partial view of the physical content provider according to our tagging mechanism as described earlier in Section 3.3.2. For the location and the phone state, we provide three different accuracy levels: *accurate, inaccurate, customized*. Option *accurate* represents the accurate location information, while *customized* indicates a customized way for replacing the raw data, such as using a fixed value or the hash value. Option *inaccurate* means partially replacing the raw data. For example, *37.422006, 37.0, 0.0* represents *accurate, inaccurate, customized* separately for *37.422006*. For camera, microphone, phone call, send/receive SMS and Internet, we give two options *yes* and *no* to indicate whether the profile supports such functionality or not. Option *no* indicates only exposing fake services instead of the real ones.

3.3.4 Profile Isolation

The Android platform facilitates component re-use to simplify app development. Besides the built-in components, apps can also provide new components for others to use. One app can take advantage of not only the built-in components in the Android framework layer, but also components of all the installed apps. It can also provide interfaces for others. For example, one app can ask a browser app to open a URL address, without being granted the network permission. The Android platform provides well-defined interfaces for the ICC. The ICC can cross the boundary of apps if it satisfies the permission checking. Based on the four basic types of Android components, ICC includes four common actions: starting an activity, binding a service, receiving a broadcast and accessing a content provider. Apps can pass an *Intent* message (an Android-specific data structure) to each other through the ICC. To ensure consistency of the virtual resource view inside each isolated environment, any cross-profile ICC is not permitted.

RVL checks whether the sending and handling components of an intent are in the same profile. It denies all the cross-profile ICCs. Specially, we treat the Android built-in apps and components as system resources, which do not belong to any specific profile. These built-in apps and components either manage sensitive resources or handle basic

Table 3.1: Resource Configuration Option

Resource Configuration $\mathcal{R}_i$	**Option**
SDcard_View	none: disallow SD card usage partial: use subfolder as root directory full: allow full access
ContentProvider_View	none: disallow all access partial: partial virtual view full: access on full content provider
Location_Accuracy	customized: customized value inaccurate: partially replaced with zero accurate: accurate value
PhoneState_Accuracy	customized: customized value inaccurate: partially replaced with zero accurate: accurate value
Camera_Support	yes: support camera no: fake camera
Microphone_Support	yes: support microphone no: fake microphone
PhoneCall_Support	yes: allow phone call no: disallow phone call
SendReceiveSMS_Support	yes: allow send/receive SMS no: disallow send/receive SMS
Internet_Support	yes: allow Internet no: disallow Internet

system events. Therefore, to have a high compatibility, any profile can communicate with the built-in apps and components, and any ICC with built-in apps and components is not treated as cross-profile. For example, the built-in *ContactsProvider* manages the contacts

information for all the profiles; the built-in *Launcher* launches an app when a user taps on the icon on the home screen; an app can use the ICC to load a URL into the built-in web browser. The sensitive resources managed by these built-in apps and components are virtually visible to all the profiles by applying the corresponding resource virtualization mechanisms as described in Section 3.3.2. In the example illustrated in Figure 3.1, the *Angry Bird* app can communicate with built-in components, such as *ContactsProvider*, but not banking apps in the other profile. With RVL, apps in one profile cannot expose any interfaces to other apps in different profiles, and therefore permission re-delegation attacks can be effectively prevented.

3.4 RVL Design

3.4.1 Architecture Overview

Figure 3.5 illustrates the overview of RVL architecture. RVL extends the Android system in both the Android middleware layer and the system call layer. The *resource virtualization layer* intercepts all the resource access and virtualizes them according to profile configurations. In the Android middleware layer, the *monitor manager* intercepts the resource access. In the system call level, the *system call interceptor* handles file-system-related and network-related system calls.

In the Android middleware layer, we add one extra built-in Android content provider, called *monitor provider*, which manages all the profile configurations, and define two new permission labels for it: `READ_MONITOR` and `WRITE_MONITOR`. We set protection level of both newly claimed permissions as `signatureOrSystem`[1] to make sure that third-party apps do not have direct read/write access to the monitor provider. We add a new system service, called *monitor manager service*, which is dedicated to communicate with

[1]Android defines four protection levels for permission labels (i.e., `normal`, `dangerous`, `signature`, `signatureOrSystem`), among which `signatureOrSystem` means that it only allows built-in apps or apps signed with the same certificate as the system image to require corresponding permissions.

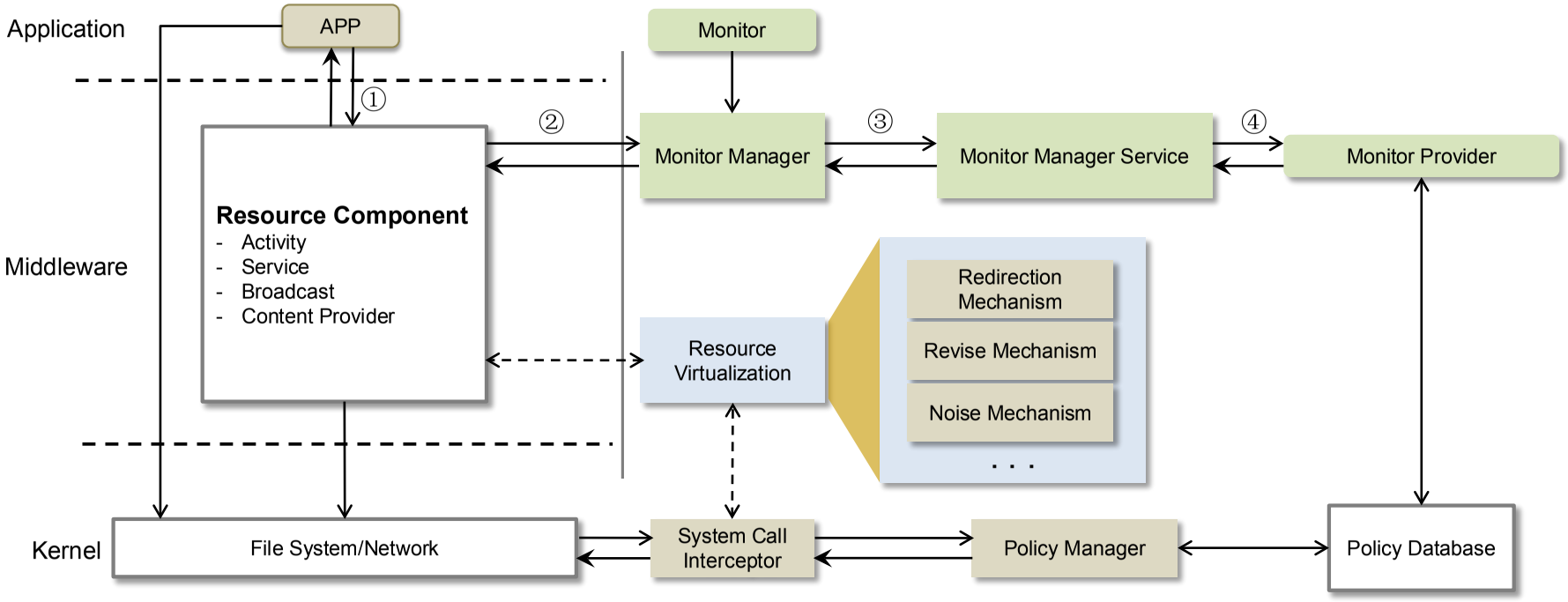


Figure 3.5: RVL Architecture

the monitor provider and provide APIs for policy management. It simply allows all read operations to the monitor provider. For write operations, it ensures that the calling app must hold the `WRITE_MONITOR` permission, which is only possible for built-in apps. We add the monitor manager which provides convenient APIs for other components to use the monitor manager service. We define the *Hook* API to insert hooks into existing components. It first collects related information of a running app before the app accesses protected resources, and then internally binds to the monitor manager service to send a query on whether this access should be allowed or denied. The monitor manager service receives the remote query issued from the monitor manager and handles it accordingly. We also add a built-in app, called *monitor*, which allows users to configure profile policies through APIs supported in the monitor manager.

In the system call level, the system call interceptor intercepts system calls and makes a decision based on policies. The *policy manager* provides APIs in C language for directly accessing policy database file.

Taking sending SMS as an example, when an app invokes the *sendTextMessage* function, it finally binds to the *ISMS* service registered by *IccSmsInterfaceManagerProxy* class and remotely calls the *sendText* function supported by *ISMS* (step 1, as shown in Figure 3.5). We insert one hook into *sendText* through the *Hook* API provided by the monitor manager. The *Hook* API collects information including package name of the calling app, the function's name (i.e., *IccSmsInterfaceManagerProxy.sendText*) and its parameters (step 2). The monitor manager further connects with the monitor manager service and remotely invokes the *getPolicy* function to query the related policy for this action (step 3). The *getPolicy* then parses the collected information and queries the monitor provider (step 4). The monitor provider manages the real policy database file and returns the policy back. After fetching the policy, *ISMS* service then selects corresponding virtualization mechanism to virtualize the resource. For sending SMS service in this case, we just make an allow/deny decision.

3.4.2 Implementation

System Call. We build a dedicated kernel module to intercept system calls and make decisions based on profile configurations. Our tool allows developers to insert custom extensions to support more advanced functionality. We monitor system calls related to the SD card and network access by Android apps. Our extension also communicates with the policy manager and virtualizes resources according to the profile configurations. To virtualize the external storage, our extension updates the *pathname* parameter of file-system-related system calls after it successfully intercepts all the system calls of monitored processes. We intercept 20 system calls which are frequently used to create/access/delete files in the SD card, including *sys_mkdir, sys_open, sys_stat64, sys_access, sys_rmdir, sys_unlink, sys_utimes*, etc. If one app accesses the SD card through any of the above system calls, we redirect the *pathname* parameter to the corresponding physical sub-folders in the SD card. Similarly for the network access, we intercept 15 related system calls, including *sys_socket, sys_bind, sys_connect, sys_listen, sys_accept*, etc., and our extension makes an allow/deny decision based on profile configurations.

Content Provider. Content Provider is one type of Android-specific components. It stores data in an SQLite database, and exposes query/insert/update/delete interfaces for other components. For example, contacts information is maintained by the built-in provider *ContactsProvider*. It uses three main tables (*contacts*, *raw_contact*, and *data*) to store all the contacts information. We add one extra field *security_tag* in each table indicating to which profile each item in the table is accessible. This field cannot be modified by apps confined in the profiles.

We intercept the contacts access in *ContentResolver*, a class that provides query/insert/update/delete interfaces for apps to access data stored inside content providers. *Uri* is one parameter of these APIs indicating which content provider should handle this request. Other parameters like *projection*, *selection* and *values* are passed to the content provider and then composed together as an SQL statement which is further handled by local C libraries. To virtualize the content provider resource, we revise the *selection* and *values*

parameters and enforce them to consider the *security_tag* field additionally. For example, in the modified *insert* API, we add *security_tag* field into the parameter *values* and then send the modified request to the content provider. We modify the built-in *Contacts* app to support modifying the *security_tag*. For *query* API, we add one more condition in the *selection* parameter related to the *security_tag* to make sure that one app can only view the item whose *security_tag* matches the profile of the calling app.

Service. Apps use APIs defined in *SMSManager*, such as *sendText*, *sendData* and *sendMultipartText*, to send SMS messages. These APIs are further handled in the *IccSmsInterfaceManagerProxy* internal service, registered as *ISMS* in the Android service manager. We modify related APIs in the *IccSmsInterfaceManagerProxy* service and enforce policies defined in profile configurations. Here we only make an allow/deny decision of sending SMS. Similar modification is applied on camera and microphone services.

Apps get location information through the *LocationManager*. We modify the *getLastKnownLocation* API which returns most recently updated location information. In this case, we directly mask the returned location information. Additionally, apps can use *requestLocationUpdates* to register a listener inside *LocationManagerService* as one *Receiver*. Then the *Receiver* will receive location updates from system drivers and notify registered apps of location update events later through the *callLocationChangedLocked* callback. We add one extra variable *mask_depth* inside the *Receiver* structure to indicate how to mask the location information later and assign *mask_depth* inside *requestLocationUpdates* during the listener registration according to profile configurations. When *Receiver* triggers the callback to notify location updates to apps, we mask the returned location values based on the *mask_depth*. Similar modification is applied on *TelephonyManager*.

Activity and Broadcast Receiver. The phone call API is not directly visible to third-party apps, so they have to delegate this task to a built-in activity through the intent `ACTION_CALL` or `ACTION_DIAL`. In the Android framework, the *Activity Manager Service (AMS)* resolves the intent messages and searches all the matching target components

to handle the requests. For SMS receiving and phone calls, apps first register one broadcast receiver with the action `android.provider.Telephony.SMS_RECEIVED` and `android.intent.action.PHONE_STATE` in the AMS. When a new SMS arrives, the AMS passes an intent message with the incoming SMS and phone call information to the registered receiver. For these two special cases, we modify the AMS to constrain the target components of the intent messages according to profile configurations.

ICC Constraint. Two Android components can communicate with each other through well-defined APIs; for example, an app can use the *startActivity* API to start a new activity. The AMS resolves the request and builds up the connection between the caller and the target components. The AMS is the centralized service for resolving all the requests across components. It provides *startActivity, bindService, broadcastIntentLocked* and *getContentProvider* APIs, each for handling one of the four basic types of ICC. Specifically, when multiple activities matches one intent, it calls the *ResolverActivity* class to resolve the intent and provide users with all the available choices. To ensure profile isolation, we intercept these APIs and only allow the ICC when the caller and the target component are within the same profile or one of them is a built-in component.

3.5 Evaluation

We implement our solution RVL in the Cyanogenmod Android version 2.3.7 and evaluate it on Samsung Galaxy Tablet GT-P1000. Our modification to the Android middleware is 658 LOC only. The modified functions are common in all the Android versions from v1.6 to v4.4, so it is easy to port RVL to other Android versions.

3.5.1 Effectiveness & Compatibility

To measure RVL's effectiveness, we selected seven apps which cover all different kinds of protected resources. These apps include *PicsArt, Voice Recorder, Wikipedia, Go SMS Pro, Call Blocker, File Manager* and *Bump*. We put them into the `Untrusted` profile,

Table 3.2: Effectiveness on Applications inside the Default Profile

Applications	**Main Functionality**	**RVL Effect**
PicsArt	taking/beautifying photos	camera area only shows black display
Voice Recorder	voice recording	record nothing during time counting
Wikipedia	surfing Wikipedia articles	showing *Article not available* page
Go SMS Pro	SMS tool	block sending and receiving SMS
Call Blocker	block unwanted phone calls	block making/receiving phone calls
File Manager	navigating SD card	only navigate inside restricted folder
Bump	social app for sharing	only a subset of contacts are accessible

a virtual device environment that virtualizes all the local resources. Table 3.2 lists the results of the resource access for each app. It shows that RVL can effectively virtualize the related resources. *PicsArt* only gets a black area in the camera display, while *Voice Recorder* actually records nothing during the time counting. *File Manager* navigates into the root directory `/mnt/sdcard` showing the path `/mnt/sdcard` in the title bar, while actually it is in the `/mnt/sdcard/Untrusted` folder. *Bump* provides an interface to share information with friends in the contacts list. It only displays the contacts with the *security_tag* `U` in *ContactsProvider*.

RVL supports multiple profiles simultaneously running on one device. Each profile has its own virtual resources, such as the external storage and user contacts; the profiles should be isolated with each other. To examine the SD card isolation, we test *File Manager* app in both `Trusted` and `Untrusted` profiles. *File Manager* app acts as if it navigates into the root directory `/mnt/sdcard` of the SD card, even though it is physically in the folder `/mnt/sdcard/Trusted` or `/mnt/sdcard/Untrusted`. According to our definition of profile isolation, apps in one profile should not be able to start any activity that belongs to other profiles. To verify profile isolation, we selected two apps *Go SMS Pro* and *PicsArt*. *Go SMS Pro* supports sending picture SMS. It shows a `Capture Picture` button, and after clicking that button, it actually prompts the user a list of all

```
Android Without RVL:
      PTID=33050001&IMEI=357453042422576&sdkver=10.7&SALESID=0006
      &IMSI=1234567890&longitude=103.77395618&latitude=1.2933376
      &DID=2001&autosdkver=10.7&CPID=3305
-----------------------------------------------------------------
Android With RVL:
      PTID=33050001&IMEI=000000000000000&sdkver=10.7
      &SALESID=0006&IMSI=1234567890&longitude=0.0&latitude=0.0
      &DID=2001&autosdkver=10.7&CPID=3305
```

Figure 3.6: RVL Effect on Geinimi Trojan

the installed apps which can handle the picture-capture action. After the user selects one from the list, the selected app takes a picture and then returns back to *Go SMS Pro*. We put *Go SMS Pro* into the `Trusted` profile, and *PicsArt* app into the `Untrusted` profile. Our experiment showed that after separating them into different profiles, *PicsArt* disappeared from the app list prompt after clicking the `Capture Picture` button in *Go SMS Pro*.

However, profile configurations sometimes may directly violate apps' main functionality. For example, blocking the network access will disable the main functionality of *Wikipedia*. We can adjust RVL to balance usability and security better. For *Wikipedia*, we can use a whitelist to only allow the network access to the *Wikipedia* server. In this case, it is safe to view *Wikipedia* articles while no communication with other remote servers can occur. *Go SMS Pro* loses a very important functionality in a virtual device without supporting sending SMS. It is difficult to differentiate whether an SMS is triggered by users or by malicious code. Users need to disable sending SMS support to block apps from making any unintended cost. However, we can still enable the receiving SMS support to balance more usability, similar with *Call Blocker*. It is flexible to adjust RVL to effectively virtualize resources without losing much app usability.

We also evaluate a malware set from [111]. Table 3.3 shows the detected malicious

Table 3.3: Malware Behavior Evaluation

Local Resources	Malware Samples
SD card	ADRD, AnserverBot, Asroot, BaseBridge, BeanBot, BgServ, CoinPirate, DroidCoupon, DroidDream, DroidDreamLight, DroidKungFu1, DroidKungFu5, FakeNetflix, FakePlayer, Geinimi, GingerMaster, GoldDream, HippoSMS, Jifake, KMin, Pjapps, RogueLemon, RogueSPPush, Walkinwat, YZHC, zHash, Zsone (27)
Contacts	DroidDreamLight, Gone60, RogueSPPush, Walkinwat (4)
Phone State	ADRD, AnserverBot, BgServ, CoinPirate, CruseWin, DroidCoupon, DroidDream, DroidDreamLight, DroidKungFu5, Geinimi, GingerMaster, Gone60, KMin, Pjapps, Plankton, RogueLemon, RogueSPPush, SndApps, Walkinwat, zHash, Zitmo (21)
Location	BgServ, Geinimi, GoldDream, Walkinwat, Zsone (5)
SMS(send/receive)	ADRD, AnserverBot, BaseBridge, BeanBot, BgServ, CoinPirate, CruseWin, DogWars, DroidDreamLight, Endofday, FakePlayer, GGTracker, GoldDream, Gone60, GPSSMSSpy, HippoSMS, Jifake, KMin, Lovetrap, Pjapps, Plankton, RogueLemon, RogueSPPush, SndApps, Walkinwat, Zitmo, Zsone (27)

behaviors inside the `Untrusted` profile. RVL can effectively prevent malware from divulging local sensitive resources. For example, *Monkey Jump 2* infected by the Geinimi trojan sends the device info and location data to remote servers every five minutes according to a stored server list. The information is used to identify the locations of all infected devices. Remote servers then send malicious commands back to remotely control the infected devices. We created a mock server to monitor the communication between an infected device and the malicious server. After we put *Monkey Jump 2* into the `Untrusted` profile, our mock server received different information. Figure 3.6 shows the decrypted data. RVL successfully virtualized the phone state and location data accord-

Table 3.4: Performance Evaluation for Various Resources

Operations	**Without RVL**	**RVL**
SD card (read)	22.90 ms	226.10 ms
SD card (write)	310.00 ms	661.60 ms
Contacts	7.83 ms	24.61 ms
Phone State	0.84 ms	18.28 ms
Location	0.43 ms	19.78 ms

ing to our configuration. With RVL, *Monkey Jump 2* only got an IMEI (the device ID for GSM phones) and a location with all the virtual values 0.0. RVL successfully protected the phone state and location information from being leaked by malware. We detected four pieces of malware accessing the user contacts and 27 pieces of malware accessing the SD card. With RVL, the `Untrusted` profile was assigned a private virtual copy of the user contacts and the SD card, so that malware was not able to corrupt the physical resources. RVL also successfully prevented 27 malware samples from either sending or receiving SMS by disabling the SMS-related service and blocking the `SMS_RECEIVED` broadcast event.

3.5.2 Performance

We build a custom app to measure the performance overhead for each type of resource access. Table 3.4 shows the results for four different types of resources. For each type of resources, we access 1000 times and calculate the average access time. In particular, we use a 1M file to measure the overhead of SD card read/write access; for the contacts access, we test a contact list containing 6 entries, and the time shown in the table is for querying the database and parsing each entry. This overhead is acceptable by most of Android apps for their low-density sensitive resource access.

To evaluate the runtime performance overhead for the whole system, we run RVL on popular Android benchmarks, AnTuTu Benchmark, Quadrant Standard Edition, RL

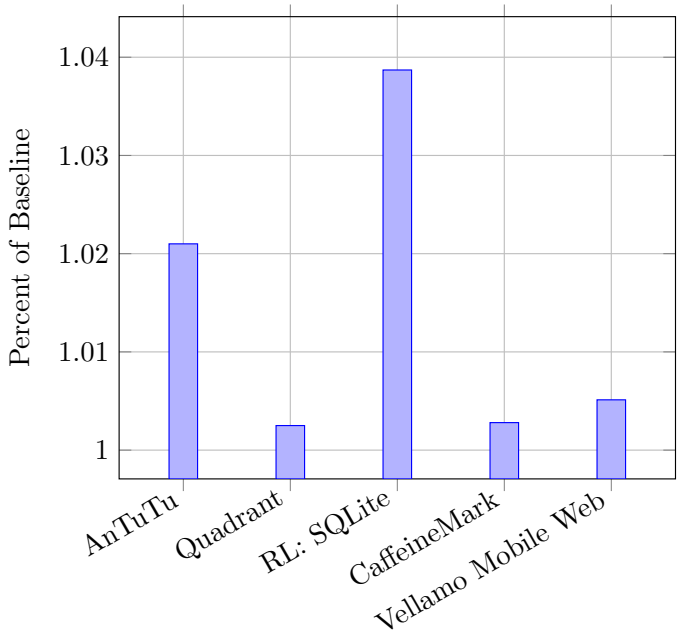


Figure 3.7: Benchmark Results

Benchmark: SQLite, CaffeineMark Benchmark and Vellamo Mobile Web Benchmark, measuring Java performance, graphics, networking, CPU/memory, database I/O, and the SD card read/write etc. The results are shown in Figure 3.7. The overhead of RVL on most benchmarks is negligibly small. The score from the RL benchmark shows a 3.87% overhead, which is mainly from the content provider I/O due to the virtualization on the content providers. However, most real-world Android apps do not involve frequent database I/O. To further improve performance, we can dispatch the centralized hook point in *ContentResolver* into multiple points in each content provider for optimization.

3.6 Related Work

Permission Control. Android uses a permission-based mechanism to restrict the capability of installed apps and protect local resources. During the installation, the package installer prompts users all the permissions required by the app. However, it is quite difficult for users to evaluate the safety only based on the permission list. Stowaway [41] has revealed that one-third of apps from the official Android market require more permissions

than what they need, and those extra permissions even confuse app developers. Meanwhile, Android does not allow users to selectively choose a subset of these permissions or revoke certain permission after the installation. Past efforts have been made to design more flexible permission systems, such as MockDroid [20] and Apex [74], which allow users to revoke granted permissions at runtime. Protecting resources through the permission system requires users to have detailed knowledge of the Android system, which is a common problem for usability. Moreover, users may make mistakes in the permission configuration. Vulnerability in ICC will also result in resource-abusing attacks. In contrast, our approach provides stronger security guarantee through isolation. It also has improved usability.

Virtual Environment. Many virtualization-based solutions have been proposed to support multiple environments at different system levels in the Android OS, such as L4Android [64], Cells [16] and AirBag [100]. Comparing with our approach, these solutions focus on building up an isolated execution environment for apps. They achieve stronger environment isolation, however, the data and services must have multiple physical copies, such as user contacts, location service and telephony service. Our solution aims to virtualize the resources only for confining resource-abusing apps, and therefore it simplifies the environment setup and incurs lower overhead considering the power and storage.

There are many solutions that perform light-weight virtualization on desktop systems, such as [67, 93, 107]. Compared to such solutions, our approach addresses challenges from virtualizing Android-specific resources, such as the content provider.

Private Data Protection in Android. Several solutions focus on preventing sensitive data from being leaked by third-party apps. For example, TaintDroid [37] is a dynamic information-flow tracking system which detects whether sensitive data are sent out through the network interface. TISSA [113] defines a privacy mode for each app, in which the Android system uses fake sensitive data to prevent untrusted apps from stealing privacy information. In contrast, the focus of our solution is to virtualize complex Android resources such as the file system and the content provider, so that it can provide a consistent

view for Android apps.

Inter-component Communication Constraint. Android provides well-defined interfaces for different components to communicate with each other. One component in one app can also interact with components belonging to another app, as long as the permission checking succeeds. Potentially through the ICC, high-privileged apps can process high-privileged tasks requested by other low-privileged ones. To detect potential transitive permission usage attacks, Woodpecker [51] statically detects capability leaks through a path-sensitive data flow analysis. Many existing solutions [75, 35, 23, 24] also apply dynamic analysis by extending the Android middleware layer to monitor the call-chain of the ICC and add additional restriction based on the ICC history. In our solution, this problem is addressed by isolation between different profiles.

3.7 Summary

As smart mobile devices integrate more resources, the physical boundary among resources disappears. Therefore critical resources are exposed to resource-abusing apps. Existing solutions fail to balance security and usability. In this work, we propose RVL which can systematically protect resources through light-weight resource virtualization. RVL defines profiles containing a set of virtual resources to confine Android apps. Profiles with virtual resources alleviate users' burden in weighing permissions requested by untrusted Android apps, and thus improve usability; isolation between profiles ensures stronger protection. We show the compatibility, effectiveness and performance through evaluation on real-world apps.

Chapter 4

DroidVault: A Trusted Data Vault for Android Devices

4.1 Introduction

The rapid adoption of mobile devices poses an imminent threat to the sensitive data in enterprises and cloud services. Mobile OSes and apps form a large, complex and vulnerability-prone software stack, which is witnessing a sharp rise in malware and OS vulnerabilities [111]. In addition, Android users often install untrusted apps or make modifications (e.g., via "rooting") to the Android OS to bypass restrictions set by vendors, thereby increasing the risk of compromising the software stack. This gives rise to a practical dilemma for data owners: should they trust users' devices and permit the use of sensitive data in devices outside their control, or should they enforce strong control over the sensitive data by banning untrusted devices. Data owners often choose to blindly trust the commodity mobile OSes and user-installed mobile apps.

Trusted Data Vault. Ideally, if mobile platforms can provide mechanisms for data owners to control the usage of the sensitive data, strong data protection can be achieved in existing mobile devices. To enable this, we introduce the concept of *trusted data vault* — a small trusted engine that data owners can trust to securely manage the storage and usage of

the sensitive data in untrusted devices. A trusted data vault must balance the misaligned incentives between data users and data owners — data users want unfettered control of the apps and the OS, while data owners need strong control over the sensitive information.

Existing work [62, 108, 99, 34] has been dedicated into building an isolated secure environment in the mobile devices. On-board credentials platform [62] designs an architecture for the credential management via a hardware-assisted secure environment. Other work [108, 99, 34] implements the Mobile Trusted Module, a secure element specified by Trusted Computing Group, through either software-based or hardware-based approaches. However, these solutions either only support limited functionality than secure storage and verification, or rely on a large trusted computing base (TCB) to perform operations on sensitive data.

Approach. In this work, we propose DroidVault, a trusted data vault for Android devices. DroidVault ensures that all the sensitive data remains encrypted throughout its lifetime in the untrusted Android device, and also supports an execution environment for trusted code to operate on the encrypted data. To extend the trust from the data owner's workspace (e.g., the enterprise workspace or cloud storage services) to the mobile client, we expose four important services in DroidVault: *secure network communication*, *secure data storage*, *secure input and output*, *secure data processing environment*.

The main challenge in designing a practical data vault is to enable limited but sufficient functionality with a minimal TCB. Existing secure hardware platforms, such as ARM TrustZone, TPM, M-shield, JavaCard and NGSCB [77], only provide limited available resources for secure environment, such as limited memory and storage, and thus to make a practical deployment, the TCB of the data vault must be kept as small as possible.

We prototype DroidVault as a small trusted hardware-assisted engine through the novel use of hardware security features supported by recent ARM CPUs, namely ARM TrustZone. It is being widely adopted in ARM-based embedded devices. Unlike software virtualization mechanisms which multiplex two executions on the same CPU, the ARM TrustZone architecture isolates the CPU core and MMU subsystem at the hardware level

and creates two environments with different security privilege levels. It supports red/green systems [63, 77, 97], which partition hardware resources into a highly-constrained trusted (green) environment and a general-purpose untrusted (red) environment. Therefore, it enables DroidVault to co-exist with a completely untrusted Android software stack. We prototype DroidVault in the trusted environment which holds a higher security privilege but only limited resources, and thus DroidVault behaves as a tiny trusted engine for handling sensitive data operations in the same device that hosts a separate untrusted Android OS.

To significantly minimize the TCB of DroidVault, we leverage the network and file system modules from the untrusted Android OS. In our implementation, DroidVault has a TCB of about 12K lines of unoptimized code — this is within the range of systems which can be formally checked by existing verification tools [61]. Note that as a prototype for now, we only use a serial console and a hardware keyboard to simulate the secure display and input. The touchscreen feature, adopted by most recent mobile phones, has already been proven practical to support the secure display and input on the ARM TrustZone architecture, and also has been demonstrated by many existing research and commercial products [9, 10, 1]. In this work, we focus more on the design of a feasible trusted data vault on top of the ARM TrustZone architecture and the analysis of its security guarantees. Thus, we simply use the serial console as our current prototype implementation and treat the touchscreen support as our future work. We have evaluated that the USB touchscreen driver in the Android source code has only 1.1K lines of code (LOC). Therefore, the TCB will not increase too much if including the touchscreen driver in the future implementation.

In summary, we claim the following contributions:

- We propose the concept of a trusted data vault and design DroidVault, a usable data vault for the Android platform. To the best of our knowledge, DroidVault is the first end-to-end platform that guarantees sensitive data protection for data owners (note that, we refer data owners to remote data-hosting servers instead of end users) in

untrusted Android devices.

- Many commercial vendors build virtualization systems on top of the ARM TrustZone architecture. In contrast, we build a red/green system without relying on virtualization, instead on partitioning. Further, previous commercial systems do not give details of their security design and implementation. Our work is the first in this aspect to our knowledge.

- DroidVault finds a sweet spot between allowing full-fledged functionality and having a small TCB for strong security. We propose a novel combination of using the ARM TrustZone primitives and reusing a large fraction of the untrusted Android stack. DroidVault has a small TCB of roughly 12K LOC (only 0.046% of the standalone Android OS).

- We evaluate the applicability of DroidVault to work on the protected data without sacrificing the data privacy and integrity. We also test the performance overhead of file downloads. The results show that it only incurs around 1s overhead to download a 10M sensitive file, which is acceptable in practice.

4.2 Overview

Consider that a user Alice who uses an Android device as a client for accessing sensitive files from trusted environments, such as her enterprise server. As an example, Alice needs to retrieve files from her enterprise server via her personal Android device, and process them on the device. In this scenario, the personal Android device, if compromised, gives malicious apps access to Alice's sensitive data. Though recent research [80, 98, 79] and commercial solutions [3, 13] have enabled protection for sensitive data using encryption, the files still need to be decrypted on the untrusted Android device. Therefore, the sensitive data is exposed in its raw form to a large and complex software stack. The data owner (note that the data owner in our example is Alice's enterprise server, not Alice) has little

control over which apps and operations can access the sensitive data.

4.2.1 Threat Model & Scope

In our threat model, the scope of our approach encompasses a broad spectrum of attacks that steal or corrupt sensitive data by exploiting vulnerabilities in the Android software stack, both at the user level and at the kernel level. Such attacks include misusing permissions [43], escalating privileges [39, 33, 43], exploiting vulnerable apps [46, 69], including malicious libraries [50], exploiting Android OS vulnerabilities [51] and compromising the Android kernel [22].

DroidVault aims to provide a trusted environment for receiving and processing sensitive files, which extends security guarantees from remote storage servers to the local Android devices. Therefore the sensitive data mentioned in this work only refers to sensitive files, excluding data in other forms, such as device attributes (e.g., GPS location and device ID).

DroidVault does not aim to protect against denial-of-service attacks or against hardware attacks. A compromised Android OS can still deny services to DroidVault or simply delete the local copy of encrypted data. DroidVault relies on a trusted execution environment that cannot be compromised in our threat model. DroidVault guarantees that only trusted code signed by the data owner can operate on the sensitive data in the trusted execution environment. However, it is out of scope if the trusted code itself behaves suspiciously, such as executing an infinite loop or intentionally leaking sensitive data publicly. To gain a strong guarantee, we prototype DroidVault via the ARM TrustZone hardware protection to defeat even threats from a compromised Android OS. However, DroidVault's integrity may be subverted by using hardware attacks (such as the Direct Memory Access attack or peripherals [27]), cold-boot attacks [53, 5] or by compromising the hardware integrity — these attacks are beyond the scope of DroidVault.

4.2.2 Trusted Data Vault

To counter a large threat landscape, we introduce the concept of a *trusted data vault*, a trusted engine that securely enables operations on sensitive data in Android devices. In our motivating example, sensitive data are decrypted before being accessed in the untrusted Android software stack. In contrast, in a trusted data vault, sensitive data are always protected with encryption techniques when the data are outside of the trusted data vault. The operations on the decrypted data can only be successfully executed inside the trusted data vault. The trusted data vault has the following security primitives:

- *Secure Data Transmission and Storage.* DroidVault provides secure network communication and secure local storage for sensitive data. Through authenticated encryption mechanisms, it guarantees the confidentiality and the integrity of sensitive data throughout their lifetime (including transmission and storage) in the untrusted OS.

- *Secure Display and Input.* DroidVault guarantees a trusted path to the end display for rendering sensitive data. Similarly, it provides a trusted path from sensitive inputs to the designated code.

- *Operations on Sensitive Data.* DroidVault only allows the authorized code to operate on the decrypted data. Any unauthorized code can only invalidate or destroy the sensitive data.

There are a few practical challenges in designing a trusted data vault in mobile devices. First, the size of the TCB in the trusted data vault should be small to be trustworthy. Second, it should be space-efficient due to resource restrictions. To minimize the size of the TCB, our trusted data vault (i.e., DroidVault) supports only limited functionality, which rules out the option of using a virtual machine to host the trusted data vault. We combine the use of new hardware partitioning primitives implemented in recent ARM CPUs (i.e., TrustZone), to support a small TCB and achieve these goals.

To protect the sensitive data, any result derived from the sensitive information inside the DroidVault cannot be leaked to the Android OS. Although this limits the functionality that DroidVault currently supports, the size of the TCB is significantly reduced. We show that DroidVault is sufficient to support several common apps which have a clear boundary between their sensitive parts and non-sensitive parts and thus can execute separately in two environments without data exchange, such as file downloading and simple document processing (described in Section 4.5). We do not provide any interface for the Android OS to retrieve any sensitive data from DroidVault. Only authorized code can be loaded into DroidVault from the Android OS and operate on the sensitive data. To minimize the size of the TCB, we restrict the sensitive data and their corresponding computational results inside the trusted data vault in our work, unless the authorized code explicitly exposes its own sensitive data to the Android OS.

4.3 DroidVault Design

We design DroidVault on the Android platform while taking advantage of the hardware-assisted isolated environment. We focus on analyzing its minimal requirements and security guarantees.

4.3.1 DroidVault Components

Figure 4.1 illustrates the design of DroidVault. To reduce the performance overhead and the size of the TCB, we choose to design DroidVault as a partitioning-based red/green system rather than a virtualization layer. DroidVault is a trusted engine which is isolated from the Android OS. It contains the following main components: the *Data Processing Module (DPM)*, the *Input/Output (I/O) module* and the *bridge module*, which are described below. DPM is the secure data processing environment and also supports secure user interaction through the I/O module. The bridge module provides interfaces for communication between DroidVault and the Android OS.

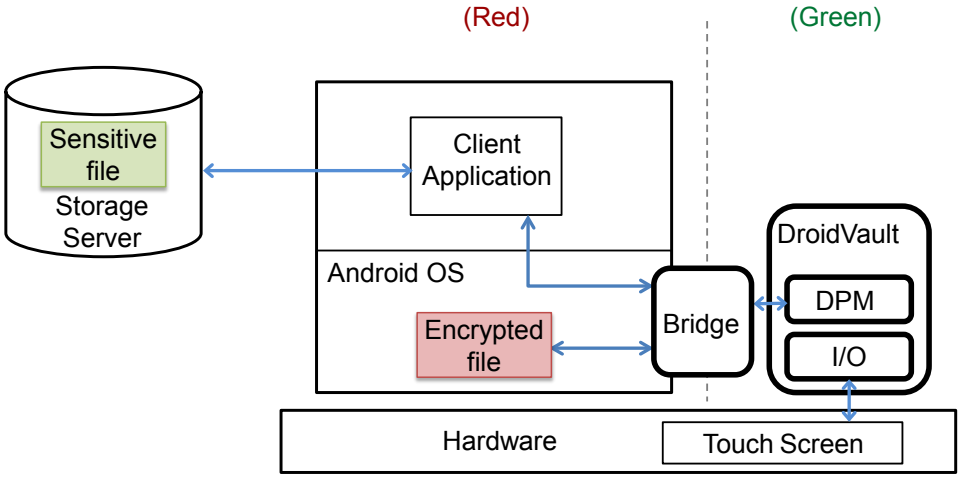


Figure 4.1: DroidVault Design

DPM. DPM is the core module for sensitive data transmission and data operations. It maintains a secure channel with the remote storage server to securely transmit sensitive data. Sensitive data are then encrypted before leaving the DroidVault environment into the Android file system. When Android apps, such as the client app in Figure 4.1, need to access the encrypted sensitive files, they must load authorized code into DPM for operations on the sensitive data. DPM module verifies whether the loaded code is signed by the data owner. Data owners take the responsibility to develop and sign the code for processing the sensitive data. DPM provides a tightly controlled runtime environment for supporting limited operations. Section 4.5 lists a few scenarios for basic data operations. DPM returns no sensitive information in plaintext to the untrusted world. The result can only be displayed inside DroidVault through the secure I/O module.

Bridge Module. To facilitate communication between DroidVault and the untrusted Android OS, DroidVault introduces the bridge module. The bridge exposes interfaces to make certain permitted function calls from one world to the other, and allows passing serializable primitive data between worlds via the shared memory. For example, the bridge module provides a single API `LoadCode` for the Android OS to load the signed code into DPM. The bridge also enables DroidVault to use resources belonging to the Android system, such as the network and the file system. DroidVault ensures that untrusted inputs cannot compromise the TCB and that the security-critical data leaving DroidVault is

encrypted.

I/O Module. The I/O module in DroidVault enables secure user input and display. DPM can request the I/O module to display sensitive data directly to users and receive user inputs.

4.3.2 Initial setup

DroidVault assumes the availability of two standard hardware primitives — secure persistent storage [96] that cannot be accessed by the untrusted Android system, and secure boot [2]. These primitives are available on existing ARM-based architectures in different ways [96, 4]. To establish trustworthy connections with authenticated servers, DroidVault stores a root certificate that identifies the root certificate authority and therefore verifies other digital certificates using a chain of trust. To prevent the untrusted Android system from masquerading as the DroidVault environment, mutual authentication is required. Data owners need to make sure that the protected files should only be received by the intended DroidVault environment. For this purpose, each DroidVault has a unique public/private key pair (K_{pub}, K_{prv}). The public key K_{pub} should be certified as a public key belonging to a compliant DroidVault system by a trusted authority, such as the device manufacturer or other trusted intermediaries (e.g., the enterprise internal certificate server). The private key K_{prv} is stored in the secure persistent storage which is only accessible inside the secure environment. We have two preparatory steps for deploying this key pair, as described below.

One Time Registration. To establish a secure channel with a remote server, a user needs to notify the data owner with his K_{pub} through one time registration. In enterprise environment, employees can register their public keys with the help of administrator. For data-hosting cloud service providers, users can log in to their accounts and then upload their public keys through particular web interfaces. Users need to contact either the administrator or the service provider if they want to change the uploaded public key in future.

Mutual Authentication. After the public key registration, mutual authentication between a remote server and a compliant DroidVault system can be achieved by following the standard client-authenticated Transport Layer Security (TLS) handshake protocol. DroidVault stores the root certificate in its secure storage and uses it to verify whether the remote server's certificate is from a trustworthy certificate authority. The remote server also authenticates the incoming connection using the public key registered by the user. The failure of mutual authentication indicates one of the following scenarios: 1) the remote server's certificate is fake; 2) the incoming connection is not from a compliant DroidVault environment; 3) the registered public key is incorrect. As to the scenarios 1) and 2), attackers cannot successfully download any sensitive data. As to the scenario 3) that attackers may change the registered public key with their own (e.g., contact the administrator by impersonating a victim), the victim can verify the registered public key by checking its hash through secure display in the secure world and make an update in time.

4.3.3 DroidVault Services

In this section, we discuss the new security services supported by DroidVault components and the security guarantees that DroidVault provides. We illustrate how the security goals stated in Section 4.2.2 are achieved.

4.3.3.1 Secure Network Communication

Due to the goal of a small TCB, DroidVault does not include the network driver in the secure world. Thus it needs to securely upload/download files to/from remote servers through the untrusted Android environment.

DroidVault supports secure communication by leveraging TLS. The corresponding data operations in TLS generally have two types: data encryption and data transmission. DroidVault handles the data encryption in the secure world, which prepares the data to be transmitted based on encryption. For this purpose, DroidVault provides different types of cryptographic APIs in the secure world, such as the symmetric cryptographic algorithm

(e.g., AES-GCM) and the asymmetric cryptographic algorithm (e.g., RSA). DroidVault holds the root certificate in its secure storage to build a chain of trust for other digital certificates and therefore authenticates remote servers.

In the data transmission phase, DroidVault requests network-related system calls (e.g., `socket`, `connect` and `gethostbyname`) from the untrusted Android OS through the bridge module. The received data from the untrusted Android OS are sanitized by DroidVault to protect against exploits from inputs. Note that the sensitive data in the network are encrypted before they leave the secure world. Therefore, the untrusted software stack in the Android OS does not threaten the confidentiality and integrity of the sensitive data.

4.3.3.2 Secure Data Storage

The secure environment only provides limited secure storage, which is not practical to store all the sensitive data. Therefore, we need to extend the secure data storage with the help of the Android file system — an untrusted but relatively large storage space. To store sensitive data in the untrusted file system, DroidVault encrypts the data and invokes file-system-related system calls through the bridge module, which include `open`, `read`, `write` and `close`. The sensitive data are in encrypted form in the untrusted Android file system. Thus DroidVault also avoids including the file system driver into its TCB.

4.3.3.3 Secure Display and Input

To provide an end-to-end channel that directly interacts with device users, DroidVault ensures that the sensitive display and input can never be accessed by untrusted Android drivers. This requires a secure overlay which securely renders any sensitive information on the screen under the complete control of DroidVault. Unlike the design for secure network communication and data storage primitives, where DroidVault can delegate most of the task to the untrusted Android OS, secure display and input must be independently supported by DroidVault through direct control over the display and input devices. We add drivers inside DroidVault to control the display and input. DPM provides the API

Table 4.1: DPM APIs

Operations	**DPM APIs**
Data Operations	`Integer Compare(Stream s1, Stream s2)`
	`Stream Concat(Stream s1, Stream s2)`
	`Integer IndexOf(Stream s1, Stream s2, Integer fromIndex)`
	`Stream SubStream(Stream s, Integer beginIndex, Integer endIndex)`
	`Stream Replace(Stream s, Stream regex, String replacement)`
	`Integer Length(Stream s)`
Network Communication	`Descriptor HttpsConnect(Stream url)`
	`Integer HttpsSend(Descriptor net, Stream s)`
	`Stream HttpsReceive(Descriptor net)`
	`Integer HttpsClose(Descriptor net)`
File System	`Descriptor FileOpen(Stream fileName, Integer mode)`
	`Stream FileRead(Descriptor file, Integer length)`
	`Integer FileSeek(Descriptor file, Integer offset, Integer whence)`
	`Integer FileWrite(Descriptor file, Stream data)`
	`Integer FileClose(Descriptor file)`
Screen Display	`Integer Display(Stream data)`
User Input	`Stream Keyboard()`

`Display` to create a secure display session, and the API `Keyboard` to receive inputs.

4.3.3.4 Secure Data Processing

DroidVault encrypts sensitive files to achieve confidentiality. To support operation on protected data, DPM is the key component, which allows the authorized code signed by data owners to operate on the sensitive data. Sensitive data are securely transmitted from a remote storage server into DPM, and then encrypted before stored in the untrusted Android OS. We use *metadata* to record the information which is used to decrypt and authenticate the encrypted sensitive data. The metadata is also encrypted and associated with the corresponding sensitive data in the Android OS.

DPM only allows the authorized code to be executed. Existing work [62, 85] has built adequate frameworks which support popular programming languages (such as Lua and C#) in an isolated secure environment for the ease of third-party development. However, considering the goal of minimizing DroidVault's TCB, it is not necessary to include a whole functional code environment into DroidVault's TCB. The size of the TCB eventually depends on the functionality to be supported. To minimize the code environment, we only provide several common functions including a set of APIs for data operations, network communication, file system access and secure display/input, listed in Table 4.1. We list the detailed description of these APIs as below.

- `Integer Compare(Stream s1, Stream s2)`

  Description Compares two streams *s1* and *s2*. If there is an index at which the two streams differ, the result is the difference between the two characters at the lowest such index. If not, but the lengths of the streams differ, the result is the difference between the two streams' lengths. If the streams are the same length and every character is the same, the result is 0.

- `Stream Concat(Stream s1, Stream s2)`

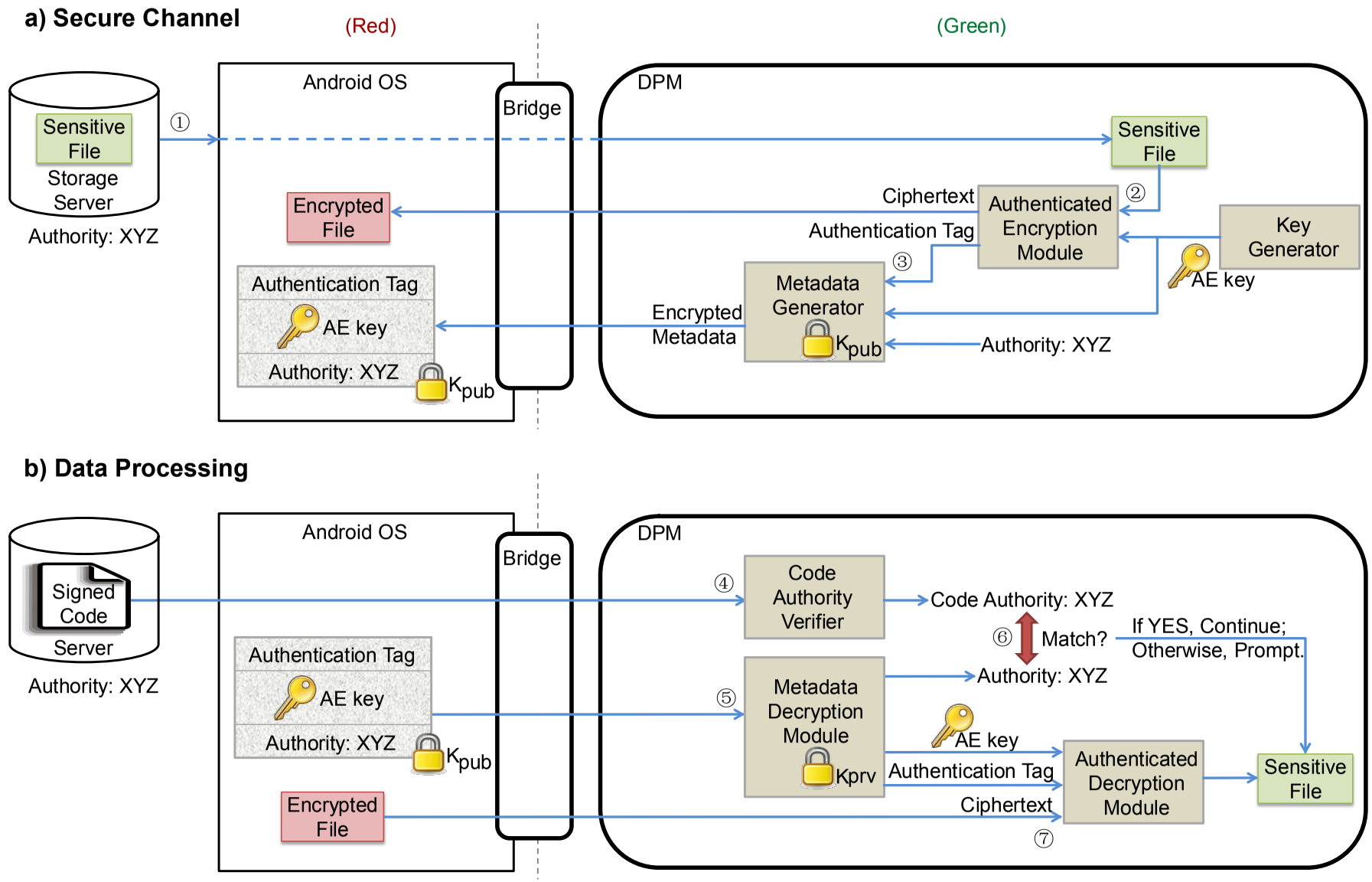


Figure 4.2: Secure Channel Establishment and Data Processing in DPM (The dash line means that the sensitive file is not directly exposed to the Android OS even though the channel goes through the Android OS)

Description Returns a new stream which is the concatenation of the stream *s1* and the stream *s2*.

- `Integer IndexOf(Stream s1, Stream s2, Integer fromIndex)`

  Description Returns the next index of the given stream *s2* in the stream *s1*, or -1. The search starts at the given offset *fromIndex* and moves towards the end of the stream *s1*.

- `Stream SubStream(Stream s, Integer beginIndex, Integer endIndex)`

  Description Returns a stream containing the given subsequence of the stream *s*.

- `Stream Replace(Stream s, Stream regex, String replacement)`

  Description Replaces all matches for the regular expression *regex* within the stream *s* with the given *replacement*.

- `Integer Length(Stream s)`

  Description Returns the number of characters in the stream *s*.

- `Descriptor HttpsConnect(Stream url)`

  Description Builds an https connection with the address *url* and returns a network file descriptor that contains details about the connection.

- `Integer HttpsSend(Descriptor net, Stream s)`

  Description Sends the data stream *s* to the network file descriptor *net*.

- `Stream HttpsReceive(Descriptor net)`

  Description Receives a data stream from the network file descriptor *net*.

- `Integer HttpsClose(Descriptor net)`

  Description Closes the network file descriptor *net* and returns the corresponding state.

- `Descriptor FileOpen(Stream fileName, Integer mode)`

Description Opens a file with the path *fileName* in the given *mode* and returns a file descriptor.

- `Stream FileRead(Descriptor file, Integer length)`

  Description Reads data of the given *length* from the file descriptor *file*.

- `Integer FileSeek(Descriptor file, Integer offset, Integer whence)`

  Description Repositions the offset of the open file associated with the file descriptor *file* to the argument *offset* according to the directive *whence*. The argument *whence* can be SEEK_SET (the offset is set to *offset* bytes), SEEK_CUR (the offset is set to its current location plus *offset* bytes), SEEK_END (the offset is set to the size of the file plus *offset* bytes).

- `Integer FileWrite(Descriptor file, Stream data)`

  Description Writes the stream *data* into the file descriptor *file* and returns the corresponding state.

- `Integer FileClose(Descriptor file)`

  Description Closes the file descriptor *file* and returns the corresponding state.

- `Integer Display(Stream data)`

  Description Displays the stream *data* and returns the corresponding state.

- `Stream Keyboard()`

  Description Pops up a keyboard and returns the user input stream.

Note that the encryption and decryption processes for secure network communication and secure data storage are transparent to the DPM code. For example, `FileRead` directly returns the plaintext of an encrypted file without requiring any further decryption in the DPM code. Any runtime environment which supports these corresponding functions can be fit into DPM (we do not argue which programming language is the most suitable

one). The DPM APIs are designed for the code running inside DPM. The authorized code is loaded from the Android OS into DPM through the bridge module API `LoadCode`. Next, we will show the details about how sensitive data are securely transmitted from a remote server to an Android OS through DPM, and how DPM processes the sensitive data.

Secure Channel. Data transmission follows successful mutual authentication (described in Section 4.3.2) between the remote server and the DroidVault execution environment. After establishing a secure channel, both sides share a secret key for further data transmission. Shown in Figure 4.2, the sensitive data are then securely transmitted from the remote server to DPM (step 1). The Android OS is not able to decrypt the sensitive data without the shared secret key even though the connection goes through its network stack (shown as dash line in Figure 4.2).

After receiving the sensitive data, DPM encrypts it and then stores it in the untrusted Android OS. *Key Generator* randomly[1] generates a key K_{AE} and sends the key as an input to *Authenticated Encryption Module*. This module encrypts the sensitive data with K_{AE} and then outputs the ciphertext and the authentication tag (step 2). The ciphertext (i.e., the encrypted sensitive data) is directly stored into the untrusted Android OS. To maintain the information which is used to decrypt the ciphertext in future, we define a metadata structure which contains the authentication tag, K_{AE} and the data authority (`XYZ` in Figure 4.2). *Metadata Generator* takes these three pieces of information as inputs to compose the metadata and then encrypts it with K_{pub}, the public key belonging to DroidVault (step 3). Therefore, the metadata can only be viewed as plaintext in DPM. The encrypted metadata is then associated with the encrypted sensitive data in the Android OS.

Data Processing. DroidVault only allows execution of the code signed by the data owner. The code can be loaded into DPM from the Android OS, or remotely downloaded from the server as Figure 4.2. The code is sent to *Code Authority Verifier* to check its integrity and authority (i.e., `XYZ`). The verifier makes sure that the code comes from the authority

[1]The random number generator can be implemented in either software or hardware manner.

`XYZ` (step 4). Before loading the encrypted sensitive data to be operated on, DPM firstly loads the encrypted metadata into *Metadata Decryption Module*. After decrypting the metadata with K_{prv}, the private key belonging to DroidVault, DPM retrieves the data's authority (step 5). DPM checks whether the data's authority matches the code's authority (step 6). Only when there is a match, it continues to load the encrypted data. *Authenticated Decryption Module* decrypts the encrypted data with the authentication tag and K_{AE} extracted from the metadata (step 7). The code can then operate on the plaintext of the sensitive file. The sensitive data are only decrypted inside DPM which is inaccessible by the untrusted Android OS.

4.3.3.5 Security Analysis

Considering our motivating example, the client on Alice's Android device loads a piece of code signed with the enterprise server's authority into DPM for downloading her document. DroidVault ensures that the document is securely downloaded from the server, marked with the enterprise authority and then locally encrypted before stored into the Android file system.

Now we give a security analysis from the perspective of data integrity and authenticity guarantees. We use metadata to maintain the keys which are used to decrypt sensitive files. The metadata is distributed into the untrusted Android OS along with the encrypted sensitive data and is only loaded when necessary. This design significantly reduces the burden of a central key management, considering the limited secure storage in the secure environment. DroidVault only needs to store an initial public/private key pair which is used to encrypt/decrypt the metadata. Each encrypted file has a corresponding piece of metadata. The mapping between the encrypted file and its metadata can be maintained in a simple way (e.g., the metadata uses the same file name with the encrypted file but with an additional suffix). If attackers corrupt this mapping, they get no benefit but the failure of file decryption.

We choose the authenticated encryption to ensure data integrity and authenticity. Con-

sidering that it is not practical to fit large volume files into DroidVault's memory, the encrypted sensitive data are read block by block for processing. The metadata is first read into memory for the authority certificate matching phase (step 6 in Figure 4.2). The encrypted file is loaded only after the matching succeeds. Attackers may replace the encrypted file by loading cipher blocks with other authorities (Time-of-check to Time-of-use attacks). DroidVault must be able to authenticate each cipher block and also identify the correct order of these blocks. Therefore, the encryption algorithm for protecting sensitive data requires a counter mode block cipher which combines both confidentiality and authenticity, such as CCM and GCM. In our work, we choose GCM for the authenticated encryption. It is possible for attackers to replace both the metadata and the encrypted sensitive data with different ones that hold the same authority as the code. In this scenario, we argue that both the data and the code belong to the same authority so that no sensitive information is leaked in plaintext unexpectedly. The code itself can identify the corresponding data to be operated on if necessary.

4.4 Implementation

We implement the DroidVault prototype in the Android Gingerbread 2.3 version on the Freescale i.MX53 Quick Start Board (QSB). We leverage the memory manager and the interrupt handler provided by Open Virtualization [9], which is an ARM TrustZone open source project currently only supporting the source code for ARM Versatile Express Board and Realview Evaluation Board. Additionally, we implement the basic execution environment for DroidVault including DPM, basic I/O, string library, encryption library, etc., and also port PolarSSL[2], a light-weight SSL/TLS library. The overall LOC is only 12,171, including the unoptimized PolarSSL library which has 8857 LOC. Comparing to the Android source code, our DroidVault's TCB is much smaller than the whole Android OS (26,710,217 LOC, 0.046%) or Dalvik (232,232 LOC, 5.24%). In this section, we

[2]PolarSSL: available at https://polarssl.org/

describe the implementation challenges when prototyping DroidVault.

Background on ARM TrustZone. ARM TrustZone is a new security extension in the ARM architecture, which has been supported since ARMv6. This feature is increasingly being utilized in emerging enterprise mobile security solutions (e.g., Samsung KNOX). The TrustZone technique is designed to support red/green systems which partition hardware resources into a secure (green) world and a normal (red) world. The two worlds are separated by hardware mechanisms, and both worlds support different privilege levels (unprivileged user level and privileged kernel level). Any interrupt can be configured to be delivered to either of the worlds, but not both. This mechanism can be used to trap all the interrupts into the secure world before they are delivered to the normal world (similar to interrupt handling in the virtualization based system [99]) or to partition the interrupt handlers (as in partitioning based systems [77, 97]). Context switches from the normal world to the secure world, which we refer to as inter-world calls, are activated through a special software interrupt generated by the `SMC` instruction. The secure world can initiate a context switch to the normal world by writing a special value to the `SCR` register. Context switches are handled by software code handlers (rather than hardware as in x86 CPUs). DroidVault handles these context switches with software handlers. The secure world can read and write arbitrary memory of the normal world, while the normal world can only operate on its assigned memory regions. This allows the secure world to build one-way memory isolation, which can be used as a mechanism to share data in the inter-world call. There are other mechanisms supported for secure boot — we do not discuss them here as these are not the focus of DroidVault's core design. Next, we describe the key techniques during prototyping DroidVault on the ARM TrustZone architecture.

World Switch. In the ARM TrustZone architecture, the `SMC` instruction is dedicated to generate a software interrupt that activates a world switch between the secure world and the normal world. `SMC` is only executable inside the kernel space with the privileged mode. It triggers the CPU to enter a special CPU mode, *Monitor Mode*, newly introduced by the ARM TrustZone architecture for interfacing two worlds. In monitor mode, we

implement the SMC handler (256 LOC), which stores all the registers of the current world and then restores the state of the other world.

The ARM microprocessor has 16 general-purpose registers (R0-R15). R0-R7 are used as either temporary registers or argument registers while the rest are preserved for other special purposes. In the ARM TrustZone architecture, all the registers are accessible in the secure world. Some privileged registers are forbidden or blanked in the normal world.

To activate an inter-world call in the kernel space, the bridge module is implemented as a loadable kernel module which adds a handler to the *ioctl* system call in the Android system. When an Android app requests services of the secure world, we use registers R0-R3 to share arguments between the two worlds. R0 and R1 are used to identify the requested service and store the return value from DroidVault to the Android OS, while R2 and R3 are used to store the information about the shared memory between the two worlds, including physical addresses of the input and the output buffer registered by the bridge and the length of each buffer.

When the secure world requests resources belonging to the Android system, we pass all the arguments to the buffer shared with the Android OS, and use registers R0 and R1 to identify the call back request and the system call type. After switching to the normal world, the bridge module takes over and handles the request from DroidVault by parsing the arguments and invoking the corresponding system call. After finishing the system call in the Android system, the bridge module writes the result back to the shared buffer and then uses the SMC instruction to switch back to DroidVault. After the world switch, DroidVault restores its previous state and continues the execution.

Porting DroidVault in Secure World. We implement the basic execution environment for the secure world on Freescale i.MX53 QSB, including the initialization code, the UART[3] driver and the interrupt handler. We support file-system-related and network-related system calls in the secure world. To reuse the Android file system and network stack for minimizing TCB, these system calls in the secure world are only wrapper inter-

[3]Universal Asynchronous Receiver/Transmitter translates data between parallel and serial forms.

faces which are further handled in the Android OS using our inter-world calling mechanism. We also port PolarSSL in the secure world. Therefore, the secure world can establish TLS channels with remote servers via the Android network stack. For the secure display and user input, we use a serial console to simulate the secure display and a hardware keyboard as the secure input device. Open Virtualization has supported the display and user input in the secure world on Samsung(R) Exynos 4412, so these two features are not fatal obstacles when porting DroidVault into the ARM TrustZone architecture. Supporting secure display on Freescale i.MX53 QSB is part of our future work.

4.5 Evaluation

In this section, we discuss the functionality and applicability of our DroidVault prototype. We integrate it with real-world apps and services. We also evaluate the performance of DroidVault.

4.5.1 New Applications Enabled by DroidVault

We successfully adapt *Dropbox* app as a cloud service provider to work with DroidVault. To evaluate the capability of the DPM module, we also develop a few applets for parsing simple documents.

Dropbox File Manager. We build a *Dropbox* file manager based on the *Dropbox* SDK to enable secure management of files on *Dropbox* using an untrusted Android device. This app allows users to securely log in to their *Dropbox* accounts, browse the *Dropbox* file system (assuming the file/directory names are not sensitive), upload/download files and search strings in the files. After receiving the user name and password from the secure input, it constructs an HTTP post message to send the password to the server and receive the response via DPM APIs `HttpsSend` and `HttpsReceive`. When a file reaches DroidVault, the file manager encrypts it in the secure world and stores it in the file system. The file manager also allows users to search a string of text in the encrypted

file. The secure world generates the `grep`-style output by invoking `IndexOf` API with the particular strings, and displays the result to users on the secure display by invoking `Display` API.

Using DroidVault services, this file manager enables secure file management and string search operation in an untrusted Android device without leaking sensitive data to the device.

Zip Archiver. Zip is a common archive file format. In the zip format, each file record is a *file entry*, which includes the file header and contents; at the end of the zip file, *central directory* contains all the offsets of these entries.

A zip parser typically follows the following steps.

1. Read a file header and check whether it is valid or not by signature matching. If so, obtain attributes of the file in the header.

2. Use the file size, the file name size and the extra data size to get file contents. Move to the next file header.

We investigate the source code of an open source *Zip Viewer*[4] (written in Java) to evaluate the feasibility of processing zip files through DroidVault. We encrypt a set of zip file samples[5] and modify *Zip Viewer* to decrypt them inside DroidVault. We develop the code running inside DPM to decrypt the files and extract the file entry names in the zip files through data operations and file-system-related operations listed in Table 4.1. It uses `FileRead` to read the plaintext of the encrypted file header and then parses the header via `Compare`, `IndexOf` and `SubStream`. We consider the file entry names as non-sensitive and thus explicitly return them back to the Android OS. We only need to rewrite 2613 LOC mainly inside *ZipInputStream.java* to request data processing in DroidVault and handle the return results. When the *Zip Viewer* starts to parse one encrypted zip file for extracting the file entry names, we intercept and load our code into DPM to decrypt

[4] *Zip Viewer*: available at http://code.google.com/p/zipviewer/

[5] These files are collected from real-world project files in Google Code: 1) guestbook_10312008 2) schema-upgrades003_019 3) google-secure-data-connector-1.2-0-bin 4) connector-otex-2.6.12-src

Table 4.2: The Performance of *Zip Viewer* when Running with DroidVault (measured in millisecond)

Projects		**1**	**2**	**3**	**4**
# of Files	/	12	29	72	162
Size of TAR	/	20K	60K	4.8M	1.0M
Size of ZIP	/	4.5K	16K	4.3M	264K
TAR	Without DroidVault	79	95	118	373
	With DroidVault	118	190.3	319.3	1037
	Overhead	49.37%	100.32%	170.59%	178.02%
ZIP	Without DroidVault	60	65	87	235
	With DroidVault	99.4	160	306.8	718
	Overhead	65.67%	146.15%	253.64%	205.53%

and parse the zip file. DroidVault returns a list of file entry names back to the Android OS and then *Zip Viewer* continues to use these results for display.

We also extend *Zip Viewer* to handle the tar format. Similarly in tar format, each file is organized as one or multiple *content blocks*, preceded by a *header block* which describes its metadata, such as the file name and the size. Each of the block has 512 bytes. Two sequential blocks filled with 0 indicate the end of a file. We only need to slightly adjust 77 LOC.

4.5.2 Performance

We build an app that downloads files of various sizes from a remote server with DroidVault. For each file, we download 1000 times and calculate the average download time. Table 4.3 shows the download time for files of different sizes.

Comparing with the normal case of file downloading, our solution has three extra steps: 1) after retrieving encrypted data from the TLS channel, the Android OS needs to copy the data into the shared memory with the secure world (Shared Memory Copy);

Table 4.3: The Performance of File Downloading inside DroidVault (measured in microsecond)

Size of File	**1K**	**10K**	**100K**	**1M**	**10M**
Without DroidVault	4084	5342	17815	148971	1335257
With DroidVault	4366	7008	33518	280805	2663760
Overhead	6.90%	31.19%	88.14%	88.50%	99.49%

2) the Android OS triggers a context switch; (Context Switch); 3) after the secure world decrypts the data in the shared memory, it encrypts it locally (Data Encryption). Note that we do not consider the decryption part as an extra step since the normal case also needs to decrypt the data retrieved from the TLS channel. To compare the weight of these three factors, we create our own micro-benchmark to measure the overhead of each step. We measure the time for shared memory copy inside the normal world and the time for data encryption inside the secure world. To evaluate the time for context switch between the two worlds, we modify DroidVault to return to the normal world without any operation inside the secure world. By running 1000 times, we get the average time for context switch around 8.4 microseconds including `SMC` interrupt and context save/restore. Table 4.4 shows our results. The main overhead comes from the context switch and data encryption. The time for context switch depends on hardware platform, which is hard to reduce. However, we can optimize it by increasing the shared memory buffer and thus reducing the number of context switches. The overhead on data encryption depends on the encryption method and the implementation. In the prototype, we have not optimized the code. The performance can be improved by several optimizations, such as adjusting the block size. It can also be significantly improved by hardware implementations [19]. As an optimization to this specific case of file downloading, we can even avoid the extra data encryption step by directly utilizing the encrypted data retrieved from the TLS channel and using the TLS session key to generate the corresponding metadata. We plan to optimize our implementation in the near future.

We also evaluate Zip Viewer to report the performance overhead introduced by DroidVault, which is incurred by encryption/decryption, context switch and data copy between the two worlds. During our experiment, we execute Zip Viewer to read archive files of various sizes in our sample set. Our result is shown in Table 4.2. Most of the overhead (50%$\sim$2x) is caused by the context switches during the interaction between the app and DroidVault. Because the overall time is small (less than 1sec), we do not perceive significant delay while interacting with the app.

4.6 Discussion

Secure Environment Indicator. It is challenging to inform users whether they are interacting with the secure display rendered by DroidVault. Overlaying the secure display on the top of the non-secure display (Gadget2008 [6]) makes it difficult for users to verify the secure display. Software indicators, such as M-GUI [91], are not suitable for DroidVault to identify the trusted environment because DroidVault needs to frequently switch with the untrusted Android OS for handling system calls (e.g., during file downloading in our motivating example). Therefore, the untrusted Android OS may completely control the GPU framebuffer during the switching and spoof the secure display. Thus we rely on hardware indicators, such as a red-green dual-color LED or a buzzer [114, 97], additionally with a time threshold `T`. DroidVault indicates a stable environment for secure UI only if there is no switch back to the untrusted Android OS for at least `T` secs. During fast switching while file downloading, DroidVault does not indicate the secure display to the user. Therefore, DroidVault does not recommend any secure UI-related actions during the file system access and network transactions. Comparing with [91, 114, 97] without allowing the switches with the untrusted domain during the secure code execution, one benefit of this design is that users do not have to wait inside the secure domain for file downloading, which gains better user experience.

Functionality Extension. In our prototype design, DPM supports only a set of primi-

Table 4.4: The Performance of Our Micro-Benchmark Test (measured in microsecond)

Size of File	**1K**	**10K**	**100K**	**1M**	**10M**
Shared Memory Copy	5	18	202	1523	11818
Context Switch	76	680	6707	67160	683347
Data Encryption	201	968	8794	63151	633338

tive APIs for minimizing TCB. We evaluate a few parsing apps and libraries for popular file formats, such as ZIP/TAR, CSV, PDF, XML and BMP. From our observation, it is sufficient to support parsing these file formats only by string comparison and delimiter searching. As one of our future work, we consider porting a small runtime environment for existing programming languages into DPM considering the ease of third-party development and the compatibility with existing third-party libraries, similar to existing work supporting Lua [62] and C# [85].

It is also challenging to extend the service supported in DroidVault while keeping the minimal TCB. In this work, we design four primitive services in DroidVault. These services are sufficient to support synchronization of sensitive documents with a remote server and data parsing and user interaction. We consider balancing between supporting functionality and the minimal TCB requirement when designing DroidVault. The design in commercial products is to run a RTOS inside the secure world, such as MobiCore [8] and PikeOS [11], with the goal of supporting rich functionality in the secure world. They usually include lots of device drivers into the secure world's TCB. However, as a goal of data protection in our work, we cut down unnecessary services to achieve minimal TCB while preserving sufficiently interesting functionality. For each selected preserved service, we further shrink the TCB as much as possible. For example, for secure network channel, we encrypt sensitive data before it goes into network channel. Therefore, it is unnecessary to include the network software stack into our TCB. For each service, we extract the core sensitive code base and delegate the rest back to the Android OS. Inevitably we need to include the drivers which directly handle the sensitive information,

such display and user input. We can also extend the secure communication channel via Bluetooth, NFC and USB similarly in the future.

We design secure display as a primitive service to meet a generic requirement of user interaction. The display driver which handles the sensitive display information must be added into DroidVault. To extend the services to support functionality provided by other device drivers such as the audio driver, these drivers which directly handle sensitive data must be also be added into DroidVault. Currently we only consider the display functionality as a primitive service.

Platform Extension. We prototype DroidVault in the ARM TrustZone architecture. However, the design of DroidVault does not rely on any specific architecture. DroidVault aims to extend the trust of data-hosting servers to untrusted clients. The design of DroidVault is extensible to other platforms. The concept of trusted data vault can be adopted widely in commodity trusted execution environment, built on top of technologies such as TPM, M-shield, Java Card, Secure Element, even the software-based hypervisor. It can even delegate the trust to a trusted third party, and be prototyped in a remote end.

Applicability Extension. DroidVault is a small trust engine designed for client devices which provides a trusted channel with data-hosting servers. Additionally, the design also fits other communication situations, such as web-based apps and mobile network communication. For example, secure SMS can be achieved by establishing a trust channel between mobile network operators and mobile clients. Banking transaction information can be securely transferred and viewed in the DroidVault through building trusted connection with web-based services. DroidVault can also be further extended to build trust between two clients (sharing secrets with friends) with the precondition of mutual authentication.

4.7 Related Work

Extending Android to Protect Sensitive Data. Several solutions extend the Android platform to protect the sensitive data. TaintDroid [37], AppFence [56], MockDroid [20], Apex [74], Saint [75], Constroid [89], TreeDroid [32], Kynoid [90], TISSA [113], Aurasium [103] and [58] enable the runtime enforcement to support rich-semantic control on sensitive data; for example, an app can specify that any other app granted the network access permission cannot read its sensitive data. Another line of research protects the sensitive data in Android by isolating the code segment according to their sources or security levels. AdDroid [76] and AdSplit [92] separate the code from different origins. They extract libraries out of the host app and use a separate process as a container to isolate them. Some existing work also protects the sensitive data by encryption. For example, CleanOS [95] is a prototype to mitigate the threat of device lost by encrypting sensitive data and evicting the encryption key to the trusted cloud in time. All above solutions in this category rely on the trust of the Android OS. In contrast, DroidVault enables the sensitive data protection in an untrusted Android system.

Virtualization on Android Devices. L4Android [64], Cells [16], AirBag [100] and TrustDroid [25] propose virtualization-based solutions to support multiple separate environments at different system levels in the Android OS. These virtualization-based solutions achieve the sensitive data protection, but the TCB is quite large, including the whole Android software stack. Even though the resources are isolated, they are still exposed to the malicious apps or the compromised OS.

Data-oriented Protection. A few abstractions are designed for data-oriented protection. Lie et al. [68] prevent memory tampering through an abstract of execution-only memory. DataSafe [30] uses memory encryption to protect data, achieving the concept of data capsules [70]. These solutions allow operations on sensitive data under the control of a policy. However, they cannot prevent information leakage through implicit flows and side channels. In contrast, DroidVault provides a stronger guarantee to secure sensitive data with a hardware-assisted isolated environment. Another related work of policy-sealed

data [86] provides a new trusted computing abstraction to protect customer data hosted by cloud services, based on that the sealed customer data can only be unsealed by nodes that match the customer-defined policy. Similar with our solution, these approaches also use encryption to protect sensitive data and only allow decryption on demand, thus reducing the potential data leakage. However, DroidVault supports richer functionality to operate on the encrypted data.

Trusted Execution Environment. On-board credentials platform [62] designs an architecture for the credential management via a hardware-assisted secure environment and provisions credential secrets that are only accessible to specific pre-authorized programs inside the secure environment. However, our DroidVault design aims to establish a secure channel with remote data-hosting servers and support secure interaction with end users, which they do not address. NGSCB [77] developed by Microsoft provides an execution environment with high isolation assurance on both software and hardware base. DroidVault can be adapted into the NGSCB architecture. Existing research has provided trusted execution environment based on the virtualization (Terra [45], Proxos [94], etc.) and trusted hardware (Flicker [72], vTPM [21], etc.). The idea is to establish trust in the system based on a small root of trust. Mobile Trusted Module is a platform-independent approach for trusted computing, similar to TPM [12]. It allows a wide range of implementations, such as based on SELinux [108] or hardware support (ARM TrustZone and Secure Element [99, 34]). However, all above solutions mainly focus on the integrity of applications. They do not preserve the application usability by allowing operations on the sensitive data. Our solution is additionally designed to support useful data operations on protected sensitive data. [114] proposes a solution of trusted path on x86 computers which establishes a protected channel between a user's I/O device and a program. Their solution is a hypervisor-based design which is claimed to be portable onto the ARM platform in the future. However, instead of building trust between a user's I/O device and a program, DroidVault aims to extend the trust with remote servers.

4.8 Summary

We present DroidVault, a trusted engine on the Android platform, to ensure the confidentiality of the sensitive data. It establishes the trust between data-hosting servers and Android devices, and provides a trusted execution environment for processing the sensitive data. We prototype DroidVault on the ARM TrustZone architecture to rigorously isolate the sensitive data from the untrusted Android OS. DroidVault has a significantly reduced TCB compared to the present Android OS. Through our evaluation, we demonstrate that DroidVault can be adopted by legacy cloud storage services and support popular operations on sensitive data.

Chapter 5

Privacy-ranking Sensitive Data Usage in Android Applications

5.1 Introduction

The Android system relies on a permission-based model to protect sensitive resources on mobile devices. However, the existing permission-based model relies heavily on users' perception of the permissions. A recent study shows that the Android permissions are insufficient for users to make correct security decisions [42]. Users have little idea about how an app would use the granted permissions. For example, to use the advertised features of an app, users may simply grant the dangerous permission to access their locations. In fact, the app may directly leak location information to an external third-party domain, or carelessly open new interfaces for other apps to escalate their privileges to access it [33]. Although several existing mechanisms have been proposed to analyze the permission usage in Android apps by detecting what and where permissions are used, they do not provide comprehensive information for users to understand how one app utilizes sensitive data after being granted permission to access. Instead, we need a solution which is both technically comprehensive and sufficiently intuitive to end users. Such a solution should help users to make wise choices to protect their privacy when they are installing

new apps.

One well-explored direction in understanding the permission usage is to apply data flow analysis on Android apps [37, 51, 101, 48, 69, 28, 87]. However, most of them only determine whether a flow to leak sensitive resources exists or not, but lack precise description regarding the internal data processing logic, i.e., whether the data usage leaks a lot of information or only a little. Thus, they are unable to inform users of the difference between an app that sends the user location to third parties, and another app that only provides a yes/no answer to whether the user is presently at a certain museum or not. Therefore, a desirable approach should deliver more insight to users regarding how their sensitive data are processed and to what extent they are leaked to other parties.

Quantitative information flow (QIF) is an emerging technique for quantifying the information leakage. Various information-theoretic metrics have been proposed, such as through one particular execution path [71] or publicly observable states [54]. Ideally, QIF could be a suitable tool to evaluate how apps use sensitive resources and how much of such information is leaked. Unfortunately, the performance and scalability of existing QIF algorithms and tools are rather limited in practice. In addition, the Android-specific event-driven paradigm heavily involves asynchronous system callbacks and user interaction, which makes it even more difficult to apply existing QIF mechanisms. Considering the huge number of Android apps and their frequent updates, we need a more efficient and scalable approach.

Our Approach. In this work, we propose a lightweight and efficient approach to ranking apps based on how they use sensitive resources. In particular, we take the location data of the mobile device as a starting point. Meanwhile, the technique is also applicable to other data types, such as the device ID and the phone number. The idea is to summarize the sequence of key operations on the location data into a *data usage pattern*, which represents the app's internal logic of the location data usage. By comparing the usage patterns by different apps, we group apps with similar functionality and rank them according to their potential leakage of the location information.

Compared to existing data flow analysis techniques that only detect the presence of sensitive data flows, we focus on identifying the important operations on the sensitive data in such flows. Specifically, we propose PatternRanker, which statically analyzes how an app utilizes the location data by analyzing its Dalvik bytecode, and extracts a general and comprehensive pattern representing the location data usage by identifying key operations on the location data. We collect all the possible operations by leveraging static program slicing and taint-based techniques, and then generate the data usage patterns through predefined heuristics (shown in Section 5.3.1). We evaluate PatternRanker on 100 top free apps that request the location permission. Our experiments show that PatternRanker effectively extracts the data usage pattern for ranking apps. PatternRanker also achieves an average analysis time of 27s per app, which is sufficiently small for analyzing real-world apps.

The applicability of the data usage pattern is not limited to app ranking. It can also efficiently assist further analysis, such as accelerating existing QIF solutions by applying their current mechanisms on our extracted patterns instead of on the raw logic of apps. It is also helpful to making suggestions on permission evolution in the Android ecosystem. It is not necessary to grant apps full access to the sensitive data in certain scenarios. For example, when an app requests the location permission, it may only want to decide whether the device is currently at home or at work. The app only needs a yes-or-no answer to satisfy its logic, instead of the raw location data. From the Android system's point of view, it is desirable to reveal as little information as possible to apps without sacrificing functionality. Based on this app's internal logic, the Android platform can support one API that only returns an identifier of the raw location and another API that calculates the distance via these identifiers. Alternatively, the app's logic can also reflect the app's requirement on the location accuracy [73], and thus the Android system can provide new location-related APIs with various accuracy levels.

To sum up, our work has the following contributions:

- We propose a lightweight and scalable approach to ranking apps' threats to user

privacy based on the usage pattern of sensitive information.

- We build a static tool to automatically analyze how Android apps utilize the sensitive data and identify the key operations.

- We evaluate a set of 100 top location-related Android apps, and demonstrate the effectiveness of our approach in ranking these apps and classifying them into different categories according to different data usage patterns.

5.2 Approach Overview

In Android, an app is required to declare a list of permissions for accessing sensitive resources. During the app's installation, the Android system prompts users with this permission list, and users must decide whether to grant all the permissions and install this app, or refuse the permissions to deny the installation. The permission-based model relies on users to make proper security decisions by examining the app's permission list. It is extremely difficult for the users without expertise to understand the precise meaning and security implication of each permission. Even if some users do have good understanding on these permissions, it is still hard to decide whether an app can be harmful, only judging from the list of requested permissions.

5.2.1 Motivating Example

Two apps both requesting the location permission can be significantly different, in terms of their internal logic of processing the location data. The internal data processing logic is the key to determining whether an app violates users' privacy. We use two real-world apps as an example to illustrate the problem.

- *GPS Share.* Users can use this app to explicitly share their current GPS locations with their friends. When users click the *Send Location* button on the main screen, the app first invokes the *getMyLocation* API to get the current location. Then it

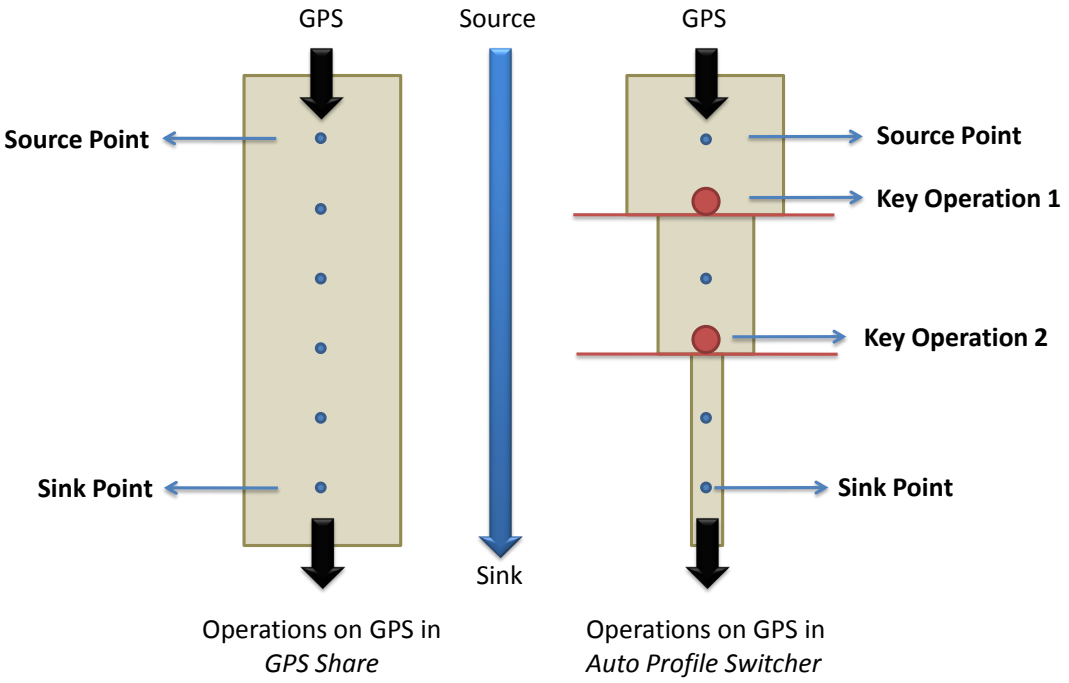


Figure 5.1: Different Operations on Location Data in Two Apps

converts the GPS coordinates into a string and performs a series of string operations, such as appending a pre-defined string. It finally sets the concatenated string as an extra field of an intent with the type of `text/plain` and passes this intent via *startActivity* to other handler apps, such as an SMS messenger app.

- *Auto Profile Switcher.* This app profiles phone states (such as ringtone, volume, silence mode and wifi) under the condition of the current GPS location. A user first pre-stores the GPS location for each profile (e.g., work and home). After launch, the app reads the current GPS via *getLastKnownLocation* and checks the closest profile by comparing the current location with pre-stored GPS locations. If the distance to the closest profile is within the pre-defined range, it automatically applies the closest profile by changing system states accordingly, such as through *setRingerMode* in AudioManager, *setWifiEnabled* in WifiManager and *vibrate*.

Apparently, these two apps have different operations on the location data. Different operations may generate different outputs that convey different amount of sensitive information. Figure 5.1 illustrates the difference between them. The width of the grey box

represents the capacity of the location data, which indicates to what degree the current output is close to the raw location data. The larger capacity means the output is closer to the original location data. In the left figure, the capacity remains the same along the flow, while in the right figure, the capacity is significantly reduced due to two key operations. The first key operation is the distance calculation between the current location with pre-stored locations, which transforms the raw location data from a accurate point to a range. The second key operation is distance comparison which further reduces the observable location-related output to one bit (true or false). The sink API (e.g., *setRingerMode*) changes the system state accordingly, which indicates one bit of information to other installed apps. Therefore, it is important to summarize these key operations as a pattern to express the sensitive data usage in one app. Traditional taint-based analysis only focuses on detecting the presence of one flow. However, we need to further focus on identifying the key operations on the sensitive data in one flow. This information can effectively assist analysts and even end users to understand the apps better and thus rank the risks of the apps.

5.2.2 Key Design Decisions

We rank one app based on data operation analysis, instead of leaking bits. The concept of *capacity*, described in our motivating example, does not reflect how many bits one operation may leak at runtime, but reflects the semantic effectiveness whether one operation preserves the raw data. For example, Android provides standard APIs *distanceBetween/distanceTo* for apps to calculate the distance between two points. However, some apps implement their own methods to complete the same task. For instance, *Auto Profile Switcher* app implements the *getMtBetweenPoints* method to calculate the distance through complex mathematical computation (including *toRadians*, *sin* and *cos*). It is extremely difficult to statically predict which set of mathematical operations may leak more bits of raw data. It is also improper to conclude that one is safer for leaking fewer bits of raw data in one particular run, because they are semantically equal even though they may

get slightly different results at runtime. Therefore, considering practicality for analyzing real-world apps, we aim to rank apps through identifying the data usage patterns, more specifically, a sequence of key operations on the sensitive data along one flow, instead of finding a metric of calculating number of bits that one app may leak.

5.3 PatternRanker Design

We aim to define a pattern to represent how an app operates on the location data, including not only a present flow from pre-defined sources to sinks, but also the key operations in the flow. The data usage pattern indicates two aspects: through which channel and to what degree the sensitive data are leaked. We use the pattern as our ranking metric. For two different usage patterns, we assign a higher rank to the one that leaks less information in the flow, and a lower rank to the one that has a simple data propagation from a source to a sink. We also consider various sink channels for ranking. For example, it gains a higher rank to share the sensitive data with a trusted service than an uncertain domain.

Assumption. The Android system provides well-defined Java interfaces for Android apps to access resources. It also supports NDK that allows developers to design their apps as native code. However, the native code is usually designed for performance improvement in CPU-intensive scenarios like game engines and physics simulation, instead of the Android-specific resource access. Hence, the native code is out of the scope in our analysis. Considering that the native code may be potentially vulnerable, we give a lower rank if any native code is included. In this section, we detail our design of the data usage pattern and a static approach to automatically extracting it.

5.3.1 Pattern Definition

We focus on analyzing the types of operations on sensitive data in a data flow. To represent how close the output of one operation is to the raw sensitive data, we attach an attribute *Capacity* to the sensitive data during their propagation. Higher value means the output

is closer to the original sensitive data. Thus we define the *Pattern* as a sequence of key bytecode operations, which aims to expressively identify the changes of the capacity in one data flow. The sensitive data enter at the source point with the maximum capacity. During the data flow, an operation may reduce the output's capacity. We also aim to use the pattern to indicate the influence of the sensitive data on the control flow, i.e., whether a code branch is conditionally triggered by the sensitive data. Thus we classify the operations into five categories: **Source**, **Sink**, **Branch**, **Capacity-preserving** and **Capacity-reducing**. Next we explain them in detail.

Source/Sink/Branch. The existing work Susi [82] has given a concrete definition for Android sources and sinks. For sources, we only consider the sources related to the location permission[1]. Additionally, the Android system supports callbacks (e.g., *onLocationChanged*) to pass sensitive data (e.g., GPS). We also consider these sensitive callbacks as sources. In addition to standard sinks, such as network APIs, we treat system state-related APIs (e.g., *setRingerMode*) and IPC channels (e.g., *startActivity*) as sinks. To avoid duplicate analysis on known trusted services and advertising libraries, we also treat these interfaces as sinks. Branch operations refer to the bytecode operations which are essential for exploring execution paths, such as `if-*` and `goto`.

Capacity-preserving/reducing. Different operations on the sensitive data may generate outputs with different capacities. According to the capacity of the output, we classify the operations into capacity-preserving (the output has the same capacity as the input) and capacity-reducing (the output has lower capacity than the input). When summarizing the operation sequence for one pattern, we ignore capacity-preserving operations because the sensitive data have only direct flow without any change of the capacity. Our goal is to identify those key operations that reduce the capacity of one flow.

Real-world apps commonly convert the data into different types or representations for programming convenience. For example, the *getMtBetweenPoints* method takes the latitude and longitude of two points as parameters, and returns the distance as the re-

[1]The mapping between the Android APIs and permissions is provided by existing tool [18].

sult, all in float (F) form. To invoke this method, the app first retrieves the raw location coordinates in double (D) form, then converts them into java/lang/String form through *java/lang/Double->toString(D)*, then into F form through *java/lang/Float->parseFloat(java/lang/String)*. During this flow, the raw value may have a loss at runtime. However, the purpose of the app is just to conveniently calculate the distance, not intentionally hide the accuracy of the location. Besides, Android provides another representation Address, which contains a street address information and can be transformed with a Location through supported geocoding and reverse geocoding. It is not reasonable to argue that it is safer to leak an Address than a Location, or it leaks less to use a float than a double or a string. All these data have the same semantics as the raw location data, even though the leaking bits do vary at runtime. We consider two scenarios of semantically equal data conversion in our static analysis: one is through specific bytecode operations of MATH type, such as *-to-* (e.g., double-to-int); the other is through specific API invocation, such as *java/lang/Double->toString(D)*, *android.location.Geocoder->getFromLocation* and *java/lang/Float->FloatValue()*.

Next, we illustrate our idea through a simple snippet of sequential operations below.

```
1  invoke-virtual {v0, v1}, Landroid/location/LocationManager;->
       getLastKnownLocation(Ljava/lang/String;)Landroid/location/Location;
2  move-result-object v2
3  ...
4  invoke-virtual {v2, v3}, Landroid/location/Location;-> distanceTo(
       Landroid/location/Location;)F
5  move-result v4
6  move v4, v5
7  cmpg-float v5, v5, v6
8  if-gtz v5, :cond_1
9  :cond_1
10 ...
11 invoke-virtual {v7, v8}, Landroid/media/AudioManager;->setRingerMode(I)
       V
```

In the above code, we can easily identify its source and sink as Line 1 and Line 11. Line 1 accesses the current location and moves it to `v2` (Line 2). Line 4 calculates the distance between the current location with another point, marked as capacity-reducing operation (CRO). The result is moved to `v4` and then to `v5`. Then Line 7 compares the distance with one value, and sets the comparison result in `v5`. Line 8 uses the comparison result as a condition to trigger a code branch that contains the sink API. The pattern for this code snippet is shown as follows.

```
1 E(SOURCE): invoke-virtual, Landroid/location/LocationManager;->
    getLastKnownLocation(Ljava/lang/String;)Landroid/location/Location;
2 E(CRO): invoke-virtual, Landroid/location/Location;-> distanceTo(
    Landroid/location/Location;)F
3 E(CRO): cmpg-float
4 E(BRANCH): if-gtz
5 E(SINK): invoke-virtual, Landroid/media/AudioManager;->setRingerMode(I)
    V
```

It is challenging to precisely distinguish whether an operation is capacity-preserving or capacity-reducing because it varies in different contexts due to two main scenarios: *Uncertain Operand* and *Uncertain Method*. Next we describe how we handle them in detail.

1) Uncertain Operand. Whether one opcode preserves the capacity depends on its operands. For example, `add-int vx,vy,vz` calculates `vy+vz` and puts the result into `vx`. Supposing `vy` is the raw sensitive data, it depends on `vz` whether the result `vx` maintains the same amount of sensitive information. If `vz` is a random number, then `vx` is not sensitive because it does not reveal the raw data `vy`. The operand may also come from an external source, such as *SharedPreferences*, external storage, network and Android specific IPC channels. It is difficult for static analysis to determine these operands, but fortunately, most above uncertain scenarios are rare to happen among the popular apps that we have studied. Only a few apps perform string concatenation and mathematical comparison operations on the sensitive data with a pre-stored default value read via

SharedPreferences API. *SharedPreferences* API is commonly used for local storage of preference settings, which we assume not to change frequently. Thus we treat them as constant values.

For some opcodes, their outputs' capacities depend on what their operands exactly are. For example, `and-int vx, vy, vz` calculates `vy AND vz` and puts the result into `vx`. In this case, even if we treat `vz` as a constant, the result `vx` is still unable to reveal `vy` unless `vz` equals to `0xFFFFFFFF`. Taking *subString* operation as another example, if it occasionally makes a substring from the beginning to the end of the string, the output string preserves the same capacity as the input string. However, these scenarios are rare to happen. Considering the semantics of the common usage of these operations, we treat them as capacity-reducing.

2) Uncertain Method. For `invoke-*` operations, we dive into the callee function to figure out its internal logic. However, if the callee is an external method that is out of our analysis code base, we are uncertain about what kinds of operations will be possibly performed on the sensitive data. In this case, we treat the external method invocation as capacity-preserving (the worse case) by default. To reduce the overestimation, we semantically identify frequently used libraries, such as String, Math and parts of Android APIs.

We list the details of our pre-defined tainting rules and operation heuristics for all opcodes as below.

CONST `const, const/*, const-*`

1. *Taint Propagation Rule.* We stop tainting the variable if a constant value overwrites it.
2. *Operation Heuristic.* None.

INVOKE `invoke-*`

1. *Taint Propagation Rule.* 1) If the method is a source API then we taint the return result; 2) Otherwise if any input is already tainted, we taint the output in the current

round and dive into the callee in the next round. Note that the input/output of one method include not only the parameters and the return value, but also the fields being accessed inside it; 3) Otherwise, if the method is an external method, we check whether it is a sink API.

2. *Operation Heuristic.* For an internal method, we treat it as capacity-preserving at the first round, and then refine it in the next round. We treat all the external methods as capacity-preserving. To reduce the false positive, we semantically mark frequently used Math/String APIs and location-related Android APIs as capacity-reducing, e.g., *android.location.Location->distanceTo/distanceBetween*.

NEW_INSTANCE `new-instance`

1. *Taint Propagation Rule.* We stop tainting the variable if a constant value overwrites it.
2. *Operation Heuristic.* None.

MOVE `move, move/*, move-*`

1. *Taint Propagation Rule.* We propagate the taint tag during `move` operations. Specially, we handle `move-result-*` operations as a pair with `invoke-*` operations.
2. *Operation Heuristic.* All are capacity-preserving operations.

BRANCH `if-*, goto, goto/*, return, return-*, packed-switch, sparse-switch`

1. *Taint Propagation Rule.* We record this branch operation if any of the involved registers is tainted. For `return` operations, we keep tracking in the caller if the return value is tainted.
2. *Operation Heuristic.* None.

GET `iget, iget-*, sget, sget-*`

1. *Taint Propagation Rule.* We propagate the taint tag from field to register.
2. *Operation Heuristic.* All are capacity-preserving operations.

PUT `iput, iput-*, sput, sput-*`

1. *Taint Propagation Rule.* We propagate the taint tag from register to field.
2. *Operation Heuristic.* All are capacity-preserving operations.

MATH `cmp-*, cmpl-*, cmpg-*, rem-*, and-*, or-*, neg-*, add-*, mul-*, div-*, rsub-*, sub-*, shl-*, *-to-*`

1. *Taint Propagation Rule.* If any of involved registers is tainted, the result is tainted.
2. *Operation Heuristic.* `cmp-*, cmpl-*, cmpg-*, rem-*, and-*, or-*, neg-*` are capacity-reducing operations. The rest are capacity-preserving operations. It is a capacity-reducing operation if it satisfies the following property: even though the sensitive operand is operated with a constant, the result value is still unable to recover the sensitive value. Take `and-int vx, vy, vz` as an example which calculates `vy AND vz` and puts the result into `vx`. Supposing that `vy` is the sensitive operand while `vz` is a constant `0x00000000`, we cannot infer `vy` given `vx` and `vz` in this scenario. `*-to-*` operations refer to the data conversion, such as `double-to-int`. Although the conversion may lose the accuracy of the raw data, it semantically behaves like a `move` operation. Thus we treat `*-to-*` as capacity-preserving operations.

ARRAY `fill-array-data, array-length, new-array, aput, aput-*, aget, aget-*`

1. *Taint Propagation Rule.* We use per-field granularity for the whole array. If a particular field of one object in the array is tainted, then the field for any object in the array is tainted.
2. *Operation Heuristic.* `array-length` is capacity-reducing and the rest are capacity-preserving.

SPECIAL `check-cast, monitor-enter, monitor-exit, instance-of, throw`

1. *Taint Propagation Rule.* None.

2. *Operation Heuristic.* None.

5.3.2 Ranking Metric

Our ranking is based on two factors: 1) through which channel the data are leaked; 2) to what degree the data are leaked. Note that one app may contain multiple patterns. Here we demonstrate the metric for ranking one pattern. We use the lowest one to represent the rank for the whole app. According to the various sinks, we classify patterns into two main categories: *In-App Usage* with a higher rank and *Sharing* with a lower rank.

For the category of in-app usage, we further classify into two subcategories: capacity-reducing pattern with a higher rank and capacity-preserving pattern with a lower rank. Considering the *Auto Profile Switcher* in our motivating example, the sink *setRingerMode* in the flow only indicates one bit of information leakage. Even if other apps exclusively monitor the state of the ringer mode in the phone, they can only infer the profile switching but no more information. However, for a capacity-preserving pattern, the sink in the flow outputs the same amount of information as the raw sensitive data. Through our analysis, we observe that some apps directly log the location-related data to the Android *LogCat*, which is publicly accessible for all the installed apps with the *READ_LOGS* permission. Some apps log the data into their local databases, which are potentially vulnerable to content pollution attacks addressed by [112]. Thus we assign a lower rank to the capacity-preserving pattern.

However, it is difficult to justify two capacity-reducing patterns, for it is improper to claim that two capacity-reducing operations (e.g., *subString*) leak less information than one. In future work, we will consider more metrics as references to compare two capacity-reducing patterns, such as how many bits of information are leaked generally in multiple

runs, which can be achieved by applying the mechanisms of existing symbolic-execution-based solutions [57, 60] on our extracted patterns to efficiently simulate multiple runs of the program and evaluate the effectiveness of these extracted key operations on the sensitive data. For now, as a first step, we only target on identifying these key operations from large-scale real-world Android apps, and thus give a rough classification while leaving further analysis as future work.

For the category of sharing, considering the scenario that it is more acceptable for users to share even the raw location data to trusted services, such as Google Map service, than to share one bit of information with untrusted domains, we further group them into three subcategories according to various sharing domains, from high rank to low rank which are *Known Trusted Services*, *Advertising Libraries* and *Uncertain Destinations*. We use a whitelist to maintain the known trusted services and advertising libraries. We give the lowest rank to those apps which transfer the location data to uncertain third party domains through network APIs, WebView APIs or IPC channels. An app may pass the location data to other installed apps through the Android IPC channels. We give it a relatively low rank due to the uncertainty about the handler app and the vulnerability (e.g., the broadcast eavesdropping risk addressed by ComDroid [31]) of the Android IPC channels. Usually, apps use dedicated libraries for common services and advertising (e.g., Google map), instead of re-implementing their own through raw Android interfaces (e.g., network APIs). The uncertain channels are mostly used to share content and resources with apps' own third party servers or can only be determined at runtime. Therefore, even though more information (e.g., the recipient's network address, phone number and package name) via these uncertain channels can be mined by applying backward analysis or dynamic instrumentation, they still fall into the uncertain subcategory. Instead, we simply categorize the sharing domains through a whitelist-based filter for common trusted services and advertising libraries collected from large-scale real-world apps.

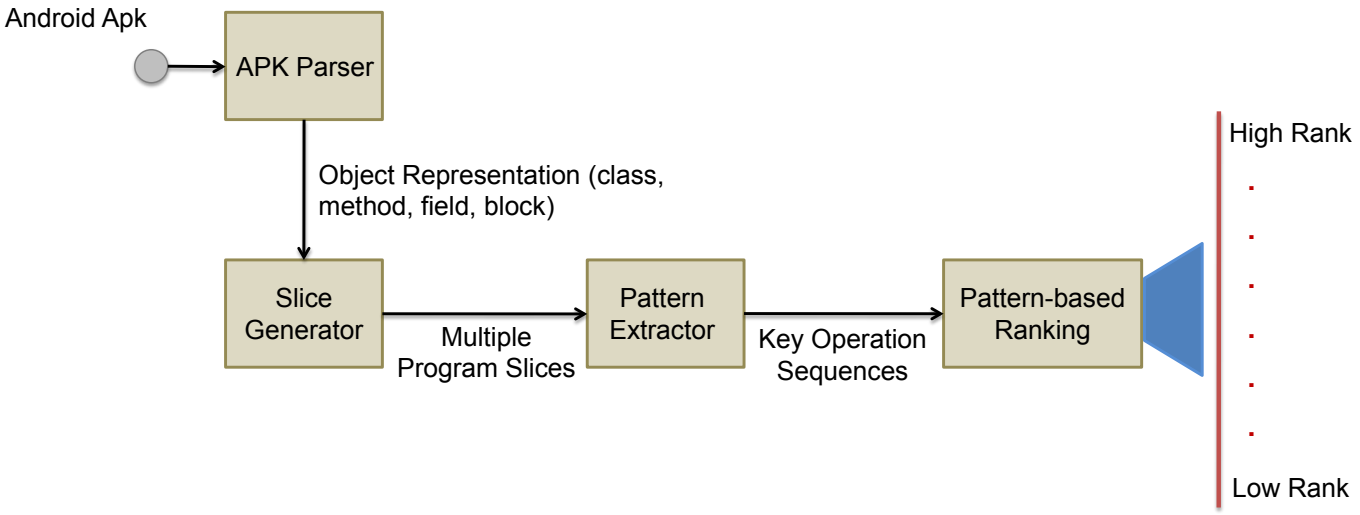


Figure 5.2: The Architecture of PatternRanker

5.3.3 PatternRanker Architecture

Figure 5.2 illustrates the overall architecture. The Android apk is parsed into appropriate object representations for further analysis, such as Smali classes, methods and fields. *Slice Generator* uses slicing technique to generate all the program slices that start from accessing the location data. *Pattern Extractor* extracts the pattern by identifying the key operations in each slice. We design a *Pattern-based Ranking* to rank the apps based on various patterns. Next we explain each component in detail.

Apk Parser. We design static analysis directly on the Android disassembled Smali code, an intermediate representation for Dalvik bytecode, which overcomes limitations of the Dalvik-to-Java bytecode transformation given concerns over the accuracy of existing translators [38]. The apk is parsed into Smali files and represented as multiple Smali classes. Each Smali class object is represented as a set of methods and fields. Each method can be further decomposed into several sequential blocks according to its internal branches. Therefore, inside one block, the instructions are sequential without any control flow. The block is treated as a minimum unit for our further analysis.

Slice Generator. We first identify the source points in the app and then perform bottom-to-top analysis. To explore all possible flow paths, we consider field-sensitive flow and Android-specific callbacks into the API hierarchy, shown in Figure 5.3. To support field-

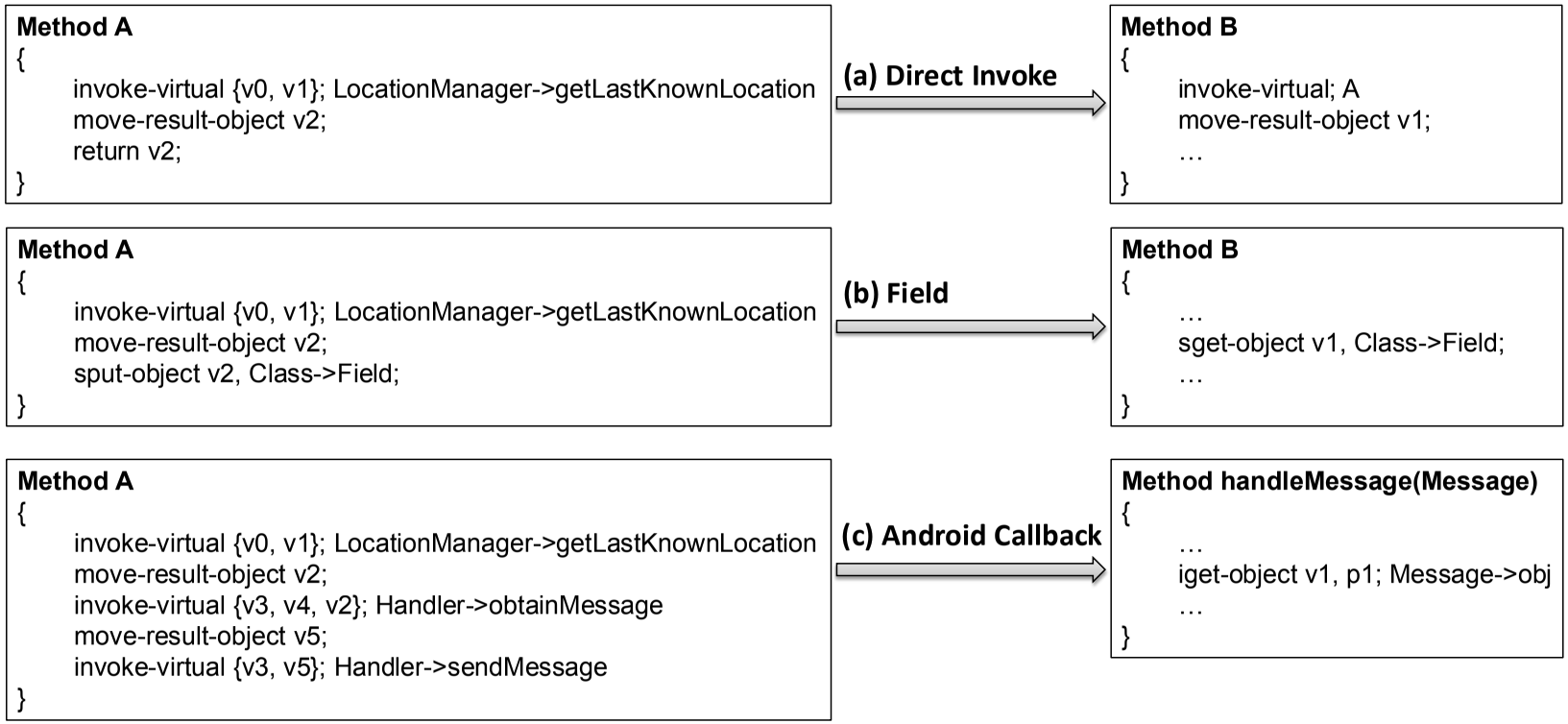


Figure 5.3: Three Scenarios for the Top Relationship in the Application Hierarchy

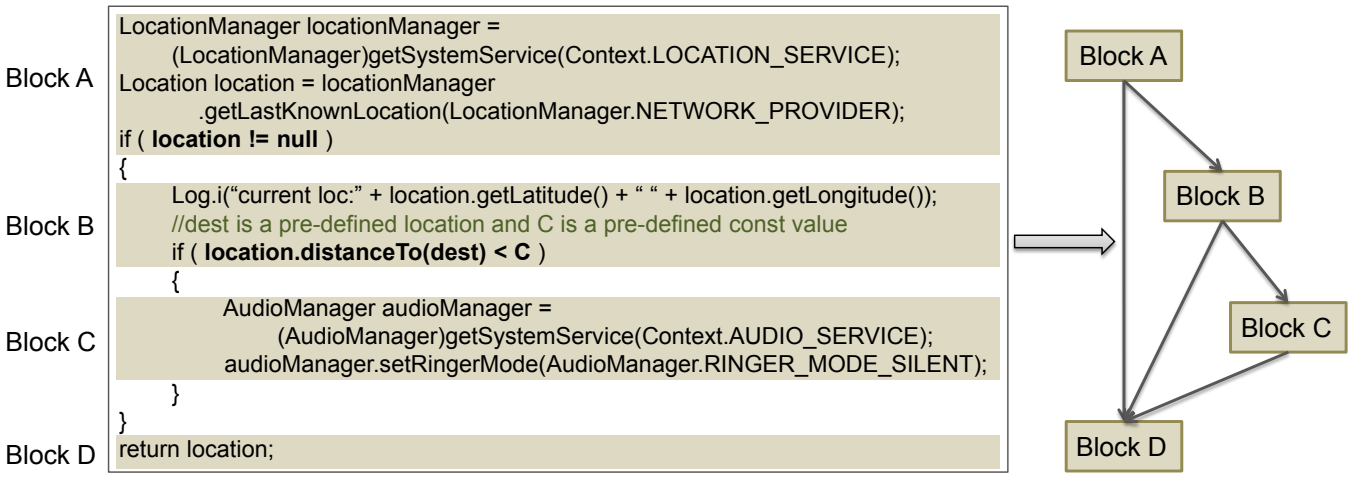


Figure 5.4: Control Flow Relationship among Blocks

sensitive flow analysis, if the sensitive data is put into one field of a Java object in one method, say *M*, we also mark all the methods reading that field as top methods of *M*. Specially, the Android-specific event-driven paradigm supports asynchronous invocation, such as *Handler*, *Thread* and *AsyncTask*. We also bridge the data flow for these scenarios. However, we do not preserve the data flow if it flows outside the app, such as file system, network and IPC channels.

From bottom to top, we analyze each method in the call chain. Intuitively, we start tracking the sensitive data from the source point and propagate the taint tags to its top methods. If one method invocation involves any tainted input, we dive into the method to figure out its internal logic. Note that to support field-sensitive analysis, the input/output of one method include not only the parameters and the return value, but also all the object fields that may be accessed inside it. For efficiency, we do not dive into any method in the publicly known libraries, such as the Android SDK, advertising/analytics libraries and other known third-party services.

Now we explain how we analyze inside one method. The method is composed of multiple blocks. We start tainting the sensitive data from the first block with the per-register and per-field granularity. At the beginning of our analysis for each block, we allocate a set of tracked registers and a set of tracked fields. Inside one block, the execution is sequential and the taint tags are propagated according to our pre-defined tainting rules (shown

in Section 5.3.1). During the tainting, we dynamically update the tracked register set and field set. After finishing one block, we go further to its next blocks. The tracked register set and field set are used as the initial sets for its next blocks. Figure 5.4 demonstrates how we handle the control flow. It may cause overtainting problem to simply taint all the next blocks if the branch condition is tainted. In Figure 5.4, we can see that the block *D* does not rely on any previous conditions, even though it is the next block of *A,B,C*. We cannot semantically conclude that the block *D* has control flow relationship with the block *B* on the condition of `location.distanceTo(dest) < C`. Thus to balance the overtainting problem, we only maintain the control flow relationship among blocks when the next block has only one reachable path, such as $A \rightarrow B$ and $B \rightarrow C$.

Pattern Extractor. After we get the program slices, we post-process them to extract patterns. As discussed in the motivating examples, the app may simply propagate the raw data until a certain point where a key operation reduces the capacity of the flow. Each slice is treated as sequential operations. We identify whether an opcode is capacity-preserving or capacity-reducing through pre-defined heuristics (shown in Section 5.3.1).

We filter out duplicate patterns if two patterns join at the same first key operation. We also semantically optimize the pattern. Certain patterns are semantically meaningless to represent the logic of operations on the sensitive data, such as commonly used null-pointer checking. Specially for the location data access, we observe that apps usually try several location providers to get the location if one fails, such as through satellites, cellular radio and network. This logic can be simplified as a single source point.

Pattern-based Ranking. As described in Section 5.3.2, our ranking system is based on two factors: through which channel and to what degree the data are leaked inside one pattern. Thus, we classify the extracted patterns by checking their sinks (indicating the leaking channel and the possible receiver) and their capacities (indicating whether one pattern contains any capacity-reducing operation). As shown in Figure 5.5, we first classify the apps into two main categories: in-app usage and sharing, by checking the sinks of the patterns. For the category of share, we further classify them into three subcate-

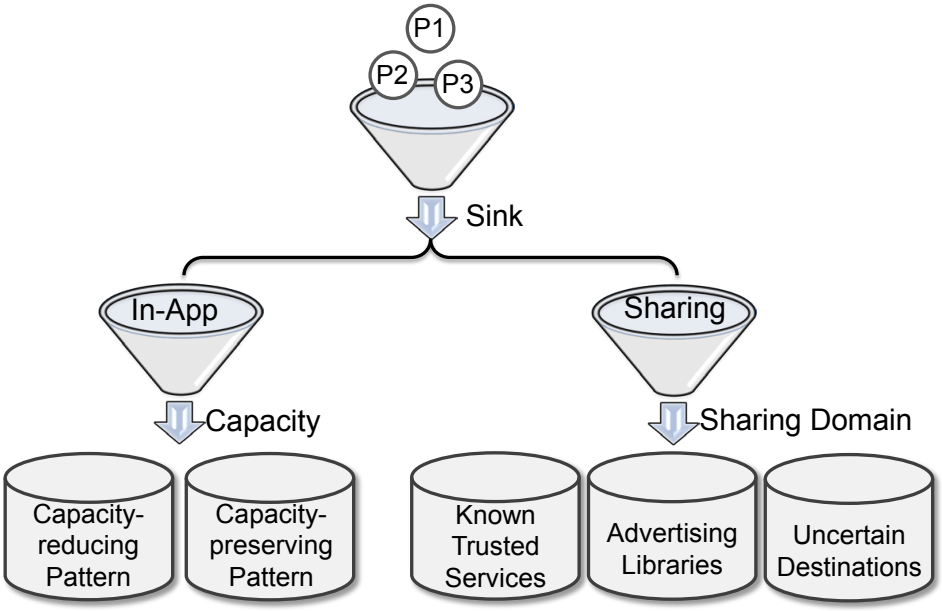


Figure 5.5: Pattern-based Ranking Schema

gories based on different sharing domains by grouping the various sinks. For the category of in-app usage, we group them into two subcategories: capacity-reducing pattern and capacity-preserving pattern, by checking whether the capacity of sensitive data at the sink point is smaller than that at the source point.

5.3.4 Discussion on False Positives

We have two types of false positives. First, the pattern extracted from our static analysis may not be triggered at runtime. For instance, we manually verified that some paths are only triggered in debug mode. Second, we may classify a capacity-reducing app into capacity-preserving category due to our conservative processing. For example, if the method is out of our analysis base (e.g., native code), we have to treat it as the worse case to avoid giving the app a high rank. However, all the above scenarios can be verified through dynamic analysis via existing instrumentation frameworks (e.g., DroidScope [105]). It is not our focus in this work and we consider it as future work.

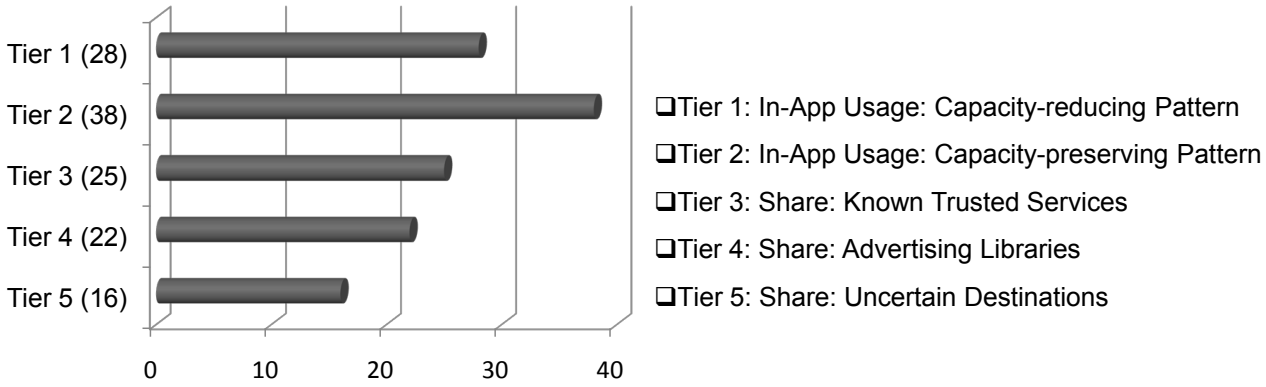


Figure 5.6: Ranking Results of Extracted Patterns

5.4 Implementation

We implement a standalone tool via Java, which directly works on disassembled Smali code. We utilize the apk parser part from the existing tool SAAF [55], which disassembles Android apks using *android-apktool* [2] and parses the smali files into appropriate object representations, such as blocks and fields. We implement the slice generator and pattern extractor. Specially, SAAF treats the `try-catch` blocks as neighboring blocks, but we do not treat this case because the `try-catch` blocks do not have logically sequential execution relationship. To avoid loops when generating the slices, we set a threshold of maximum recursive depth for block analysis. Additionally, we optimize the path by avoiding duplicated blocks. We also avoid analyzing duplicated paths if the current block can be concatenated to an existing path.

5.5 Evaluation

We collected 100 top location-related Android apps from the official Android market (i.e., Google Play) as our sample set, and ran our PatternRanker prototype on the sample set in a Debian system on a server of Intel Xeon E5-2640@2.50GHz with 64G memory. Next, we show our evaluation results in detail.

[2]android-apktool: available at https://code.google.com/p/android-apktool/

Table 5.1: Different Categories of Sharing Domains

Known Trusted Services		**Advertising Libraries**		
Aponia Map (4)	MapQuest Map (1)	adfonic (1)	google/ads (1)	mopub (2)
Geolife SDK (1)	Yandex Map (1)	admob (2)	inmobi (1)	revmob (1)
Google Apps API (1)		adwhirl (1)	imapp (1)	scringo (1)
Google Map (24)		afnn (3)	jumptap (1)	sellaring (1)
GPS-DFCI (1)		daum (2)	madvertise (1)	smaato (2)
LuckyCatLabs Sunrise/Sunset API (1)		flurry (5)	mobfox (2)	youmi (1)

5.5.1 Application Analysis on Location Usage

Specifically, we found that 28 apps of them have capacity-reducing patterns for in-app usage. Sharing is an appealing feature on the mobile platform, especially as the social networking becomes popular. We observed that a common usage for location data is to share the raw location data with trusted services and advertising libraries. According to our ranking design, we list them in the following categories from high rank to low rank, shown as Figure 5.6. One app may include multiple patterns. Here the statistics for each category shows the number of apps having that pattern.

1) In-App Usage: Capacity-reducing Pattern. We identified 28 apps that have capacity-reducing patterns. We observed that 15 of them do the distance calculation operation, among which 10 use the default *distanceTo/distanceBetween* Android APIs and the rest 5 implemented distance calculation by themselves using Math libraries. Supposing the Android system provides new location-related APIs that do not require the location permission, i.e., *(int)getLocation* returning an identifier of the location and *(float)distanceBetween(int id1, int id2)* calculating the distance via two location identifiers, these apps can benefit from it and remove the location permission from their permission lists following the least privilege principle. Next we take two examples to demonstrate the extracted patterns from them.

Auto Profile Switcher app, with more than 1,000 downloads, allows users to configure several profiles, such as *home* and *work*. It automatically switches the profile based on the current location. For example, it changes the ringer mode accordingly if the current location is within the range of a predefined area. The extracted pattern for it is shown below. First, they implement the distance calculation through the math library (e.g., `atan2` by themselves and then make a comparison operation. Through manual analysis, we observe that `cmpg-float` opcode compares the calculated distance with a value read from local storage implemented as *SharedPreferences*. This information can be obtained automatically by applying backward slicing technique [55] on key operands, which we consider it as future work.

```
1 E(SOURCE): invoke-virtual, Landroid/location/LocationManager;->
    getLastKnownLocation(Ljava/lang/String;)Landroid/location/Location;
2 E(CRO): invoke-virtual, Landroid/location/Location;->getLatitude()D
3 E(CRO): invoke-virtual, Landroid/location/Location;->getLongitude()D
4 E(CRO): invoke-static/range, Ljava/lang/Math;->toRadians(D)D
5 E(CRO): ...
6 E(CRO): invoke-static/range, Ljava/lang/Math;->atan2(DD)D
7 E(CRO): cmpg-float
8 E(BRANCH): if-gtz
9 E(SINK): invoke-virtual, Landroid/media/AudioManager;->setRingerMode(I)
    V
```

Another pattern example shows the internal logic of how the GPS data affect the display. *GPS Speedometer* is a functional speedometer app with 50,000 - 100,000 downloads. It displays the location-related information and users' travel histories on the screen. One capacity-reducing pattern extracted from this app is shown below. It reads the bearing information from the location data. The sink is an Android-specific display API *setText*. The pattern shows that the GPS data go through multiple comparisons and branch operations to finally decide the string to be displayed. From the final output display, we can only infer a rough movement direction instead of the exact bearing information. Through

manual analysis, we observe that the app compares the current bearing with a set of constant values and then decides to display a string from `N,NE,E,ES,S,SW,W,NW`.

```
1 E(SOURCE): Lcom/ape/apps/speedometer/SpeedometerMain;->
    onLocationChanged(Landroid/location/Location;)
2 E(CRO): invoke-virtual, Landroid/location/Location;->getBearing()F
3 E(CRO): cmpl-double
4 E(BRANCH): if-gez
5 E(CRO): cmpg-double
6 E(CRO): ...
7 E(BRANCH): if-gez
8 E(SINK): invoke-virtual, Landroid/widget/TextView;-> setText(
    Ljava/lang/CharSequence;)V
```

2) In-App Usage: Capacity-preserving Pattern. Information display is one important feature for apps to rich their functionality and convenient users. 25 apps displayed GPS-related information, such as position and signal strength of satellites, accuracy, speed, acceleration and altitude. We observed the following UI-related sinks: *TextView(19), Canvas(7), Toast(3), RemoteViews(2), EditText(1), Notification(1).*

30 apps logged the GPS data locally. 13 of them directly output the data to the `LogCat`, which is a public channel for all the installed apps with the `READ_LOGS` permission. However, through manual analysis on the path condition, we found most of them only logged the GPS data in debug mode. This can be verified through the dynamic instrumentation framework [105]. We also observe other logging channels: *database(10), Bundle(4), Java/IO(8)* and *SharedPreferences(6).*

3) Sharing: Known Trusted Services. From our statistic in the category of known trusted services, 25 apps used Google map service while the rest 9 used other third-party services, shown in Table 5.1.

4) Sharing: Advertising Libraries. In-app advertising has become an important revenue-generating model for mobile apps. Many of them requested the GPS location for providing targeted advertisement. We observed 22 apps include advertising/analytics libraries

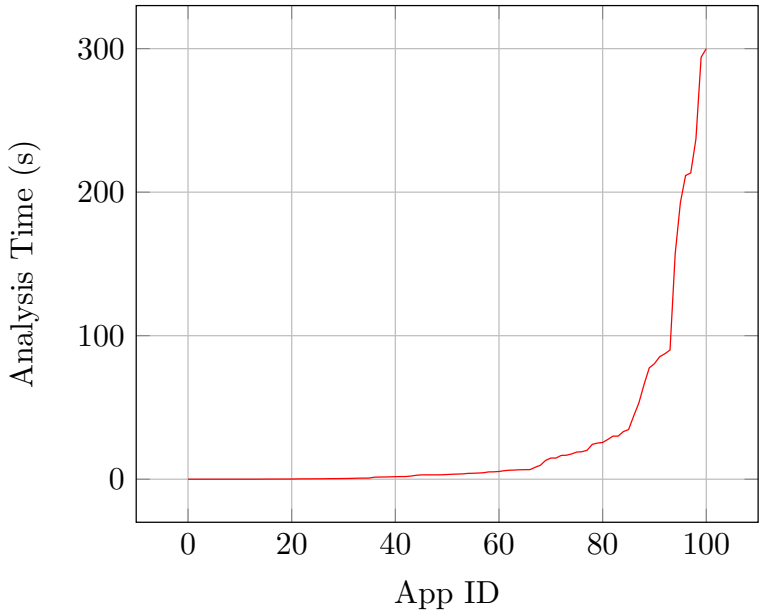


Figure 5.7: Analysis Time Distribution

potentially accessing the location data from 18 different advertising providers, shown in Table 5.1. Commonly one app includes multiple advertising libraries to increase its ads revenue.

5) Sharing: Uncertain Destinations. We observed that 8 apps share the GPS via IPC channels. Detecting intent handlers is not our focus, which can be achieved by existing instrumentation frameworks [75, 31]. We temporarily marked the intent handlers as uncertain destinations. We also observed the following sinks: *org/apache/*/HttpPost (5)*, *java/net/DatagramSocket (1)*, *WebView→loadDataWithBaseURL (1)* and *sendTextMessage (1)*. We manually analyzed the WebView and SMS scenarios. In the app *GPS QIBLA LOCATOR*, it uses the WebView API to load an iframe with a URL composed of a fixed third party domain and the geolocation as the parameters. *Mobile Chase-GPS Tracker* registers an *onLocationChanged* listener, inside which it composes an SMS message using the location information and sends it to a phone number stored in the *SharedPreferences*.

5.5.2 Analysis Time

Figure 5.7 shows the distribution of analysis time for all the apps. Note that the analysis time excludes the apk parsing phase handled by the existing tool SAAF [55]. The average of analysis time is about 27s per app. 35% of apps finished within 1 sec, due to the simple flow of the GPS data in them. Within one minute, we achieved 87% coverage. Comparing with other tools [17, 106] at the scale of minutes or larger for real-world apps, the average analysis time of our approach is sufficiently small for ranking a large number Android apps.

5.6 Related Work

In this section, we discuss about recent related work in Android regarding the permission-based model enhancement and the information flow analysis.

5.6.1 Permission Use Analysis

Android uses a permission-based model to restrict the capability of installed apps and protect local resources. During installation time, the package installer prompts users with all the permissions required by the installed app. However, it is quite difficult for users to evaluate the safety only based on the permission list. Existing solutions, such as Mockdroid [20] and Apex [74], provide flexible permission manager systems which allow users to dynamically select and revoke permissions. There are lots of research on the capability leakage problem through static detection, such as Woodpecker [51], SEFA [101], DroidChecker [28] and PermissionFlow [87], and dynamic monitoring by extending the Android framework, such as Saint [75], QUIRE [35], XmanDroid [23] and [24]. They enhance the permission-based model by extending the existing Android framework. However, different from them, our goal is to analyze the permission use in one app from the perspective of data usage.

Stowaway [41] reveals the permission bloat problem, by building the map between

the APIs to the Android permissions and then statically detecting the API coverage for large-scale apps in the official Android market. It shows that one-third of apps from the official Android market request more permissions than what they need, and those extra permissions even confuse app developers. VetDroid [109] builds a dynamic framework to construct sensitive behaviors by monitoring what and where permissions are used in the app. Pegasus [29] statically represents the interplay between the Android event system and the permissions in an app, and enforces new permission use polices on event sequences. Permlyzer [104] proposes a general-purpose framework for dynamically analyzing the permission use sequence. They evaluate not only where the permission is used, but also the cause and the purpose by checking the permission invocation sequence (e.g., a LOCATION-INTERNET sequence indicates a potential location leakage). The above permission use analysis more focuses on permission invocation, such as where and under what condition one permission is invoked. In this work, we focus on detecting what operations are performed on the sensitive data protected by one permission, which is more comprehensive for users to understand the potential behavior related with their privacy in one app.

5.6.2 Privacy Leakage Detection

Several solutions focus on preventing sensitive data from being leaked by third-party apps. Many static taint-based techniques are applied for privacy leakage detection, such as FlowDroid [17], ScanDroid [44], AndroidLeaks [48], CHEX [69]. However, they only limit to detect whether there exists one data flow from pre-defined sources to sinks. Our approach focuses more on what operations are performed along the path and identifies the key operations on the path.

Another line of work for privacy leakage detection is the symbolic execution technique. For example, SymDroid [57] designs a symbolic executor based on their defined simple version of Dalvik VM, i.e., μ-Dalvik. Similarly, ScanDal [60] designs an intermediate language, called Dalvik Core, and collects all the program states during the ex-

ecution of the program for all the inputs. Considering the Android-specific event-driven paradigm, AppIntent [106] proposes a more efficient event-space constraint guided symbolic execution. These symbolic execution based approaches aim to precisely detect path conditions for a flow. However, it is usually time-consuming for full-fledged analysis to explore all the feasible paths in one app. In our approach, we do not need any interpreter to maintain the program state, like what symbolic execution based approaches do. Instead, we only perform a lightweight slicing analysis on the sensitive data and identify the key operations that we are interested at.

5.6.3 Quantitative Information Flow

Various information-theoretic metrics have been proposed in the QIF area. For example, one particular execution path conveys certain amount of sensitive information. Flowcheck [71] is a dynamic analysis tool for Linux executables, which uses a graph-based analysis to approximate the total number of different public outputs that one program can produce. Jonathan et al. [54] build a static tool for quantifying the leakage of system software based on bounded symbolic model checking (CBMC). They aim to model the input/output behavior of a C function. However, Android is an event-driven system and has multiple entry points for one app. It is hard to precisely decide how much one particular execution path may convey and model all the behaviors for an event-driven system. Therefore, we do not focus on deciding how much information one path itself conveys through observing possible program states. Instead, the data usage pattern, we provide in this work, represents the key operations inside one particular path and indicates the degree of data leakage after performing these key operations.

5.7 Summary

To provide users with more intuitive measures to understand how apps treat their privacy, we build a tool PatternRanker to automatically extract the data usage pattern to express it.

Comparing to existing taint-based techniques that focus on detecting the presence of one flow, our approach effectively identifies the key operations in the flow. Our experiments on the real-world apps demonstrate its effectiveness and efficiency for ranking a large number of apps.

Chapter 6

Conclusion

As Android devices are becoming increasingly popular and broadly involved in many areas, such as on-line banking and social networking, a secure and reliable mobile device is an urgent need for sensitive resources. The default permission-based protection in Android requires users to estimate the risks of an app and then assign proper capabilities to an unknown app, which is a concern in both effectiveness and usability in practice to satisfy a diverse demand for protecting various types of resources.

To mitigate the threats to sensitive system resources in Android from untrusted apps, we propose and implement RVL, which spawns a virtual environment for various Android resources, including premium services and user privacy. RVL shadows physical resources through resource virtualization, thus preventing untrusted third-party apps and even malware from abusing these resources. RVL effectively confines apps' potential threats into their own virtual environment. Additionally, it is highly compatible with existing Android apps.

To further provide tight control over the usage of more important resources, such as user credentials, we propose DroidVault that separates sensitive data and operations into a trusted standalone environment. In this environment, we ensure that only authorized apps can perform operations on sensitive data. We have implemented DroidVault prototype on the ARM TrustZone architecture, and evaluated its applicability with legacy real-world

apps.

Finally, to provide better understanding on sensitive data usage by real-world apps, we design PatternRanker which statically analyzes how Android apps utilize sensitive data, specially, the impact of a sequence of operations on the sensitive data. It informs end users regarding through which channel and to what extent the sensitive data are leaked by an app, and thus helps them to make wise decisions to protect their privacy. Our evaluation on real-world apps demonstrates its effectiveness and efficiency on ranking the risks on location data for a large number of Android apps.

In summary, we design new resource-centric frameworks for different levels of security guarantees, and provide system mechanisms for resource-centric protection and information analysis on the Android platform.

Bibliography

[1] An Exploration of ARM TrustZone Technology. http://genode.org/documentation/articles/trustzone.

[2] ARM Security Technology: Building a Secure System using TrustZone Technology. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/DABGFFIC.html.

[3] BoxCryptor. https://www.boxcryptor.com/.

[4] CryptoCell for TrustZone: Comprehensive Security Sub-system for Application Processors with TrustZone. http://www.discretix.com/cryptocell-for-trustzone/.

[5] Danger on ice: Android info thaws in cold boot attack. http://phys.org/news/2013-02-danger-ice-android-info-cold.html.

[6] Gadget2008 product design. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/ch06s03s03.html.

[7] Lookout Mobile Security. http://blog.mylookout.com/blog/2011/01/07/geinimi-trojan-technical-analysis/.

[8] MobiCore. http://www.gi-de.com/gd_media/media/en/press/prs_1/pdf_2012/SamsungGalaxyS3_MobiCore.pdf.

[9] Open Virtualization. http://www.openvirtualization.org/.

[10] Qualcomm Security Solutions. https://www.qualcomm.com/products/snapdragon/security.

[11] SYSGO Demonstrates PikeOS and Android Running ARMs TrustZone. http://www.sysgo.com/news-events/press/press/details/article/sysgo-demonstrates-pikeosTM-and-androidTM-running-arms-trustzoneR/.

[12] Trusted Computing Group - Trusted Platform Module. http://www.trustedcomputinggroup.org/developers/trusted_platform_module.

[13] Viivo: Cloud File Encryption. http://viivo.com/.

[14] Android developers. http://developer.android.com.

[15] Androlib. http://www.androlib.com.

[16] Jeremy Andrus, Christoffer Dall, Alexander Van't Hof, Oren Laadan, and Jason Nieh. Cells: A Virtual Mobile Smartphone Architecture. In *Proceedings of the 23rd ACM Symposium on Operating System Principles*, SOSP '11, 2011.

[17] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, 2014.

[18] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, 2012.

[19] T.Ravichandra Babu, K.V.V.S.Murthy, and G.Sunil. AES Algorithm Implementation using ARM Processor. In *IJCA Proceedings on International Conference and workshop on Emerging Trends in Technology*, ICWET '11, 2011.

[20] Alastair R. Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. MockDroid: Trading Privacy for Application Functionality on Smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11, 2011.

[21] Stefan Berger, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. vTPM: Virtualizing the Trusted Platform Module. In *Proceedings of the 15th Conference on USENIX Security Symposium*, SEC '06, 2006.

[22] Jeffrey Bickford, Ryan O'Hare, Arati Baliga, Vinod Ganapathy, and Liviu Iftode. Rootkits on Smart Phones: Attacks, Implications and Opportunities. In *Proceedings of the 11th Workshop on Mobile Computing Systems and Applications*, HotMobile '10, 2010.

[23] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks. Technical Report TR-2011-04, 2011.

[24] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards Taming Privilege-Escalation Attacks on Android. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, NDSS '12, 2012.

[25] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Ahmad-Reza Sadeghi, and Bhargava 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, 2011.

[26] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: Behavior-based Malware Detection System for Android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, 2011.

[27] Brian Carrier and Joe Grand. A Hardware-based Memory Acquisition Procedure for Digital Investigations. *Digital Investigation*, 1(1), 2004.

[28] Patrick P.F. Chan, Lucas C.K. Hui, and S. M. Yiu. DroidChecker: Analyzing Android Applications for Capability Leak. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, 2012.

[29] Kevin Zhijie Chen, Noah Johnson, Vijay D'Silva, Shuaifu Dai, Kyle MacNamara, Tom Magrino, Edward XueJun Wu, Martin Rinard, and Dawn Song. Contextual Policy Enforcement in Android Applications with Permission Event Graphs. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium*, NDSS '13, 2013.

[30] Yu-Yuan Chen, Pramod A. Jamkhedkar, and Ruby B. Lee. A Software-hardware Architecture for Self-protecting Data. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, 2012.

[31] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing Inter-application Communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, 2011.

[32] Mads Dam, Gurvan Le Guernic, and Andreas Lundblad. TreeDroid: A Tree Automaton Based Approach to Enforcing Data Processing Policies. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, 2012.

[33] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege Escalation Attacks on Android. In *Proceedings of the 13th International Conference on Information Security*, ISC '10, 2011.

[34] Kurt Dietrich and Johannes Winter. Towards Customizable, Application Specific Mobile Trusted Modules. In *Proceedings of the Fifth ACM Workshop on Scalable Trusted Computing*, STC '10, 2010.

[35] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. Quire: Lightweight Provenance for Smart Phone Operating Systems. In *Proceedings of the 20th Conference on USENIX Security Symposium*, SEC '11, 2011.

[36] Egham. Gartner Says Annual Smartphone Sales Surpassed Sales of Feature Phones for the First Time in 2013. `http://www.gartner.com/newsroom/id/2665715`. Gartner Newsroom. Retrieved 2014-02-13.

[37] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol 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, 2010.

[38] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A Study of Android Application Security. In *Proceedings of the 20th Conference on USENIX Security Symposium*, SEC '11, 2011.

[39] William Enck, Machigar Ongtang, and Patrick Mcdaniel. Mitigating Android Software Misuse Before It Happens. Technical Report NAS-TR-0094-2008, 2008.

[40] William Enck, Machigar Ongtang, and Patrick McDaniel. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, 2009.

[41] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android Permissions Demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, 2011.

[42] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android Permissions: User Attention, Comprehension, and Behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, 2012.

[43] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. Permission Re-delegation: Attacks and Defenses. In *Proceedings of the 20th Conference on USENIX Security Symposium*, SEC '11, 2011.

[44] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. SCanDroid: Automated Security Certification of Android Applications. *Manuscript, Univ. of Maryland*, 2009.

[45] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A Virtual Machine-based Platform for Trusted Computing. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, 2003.

[46] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, 2012.

[47] Kostas Anagnostakis Georgios Portokalidis, Philip Homburg and Herbert Bos. Paranoid Android: Versatile Protection For Smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, 2010.

[48] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a

Large Scale. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, TRUST '12, 2012.

[49] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker. In *Proceedings of the 6th Conference on USENIX Security Symposium*, SEC '96, 1996.

[50] Michael Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe Exposure Analysis of Mobile In-App Advertisements. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, 2012.

[51] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, NDSS '12, 2012.

[52] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, 2012.

[53] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Cal, Ariel J. Feldman, and Edward W. Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *Proceedings of the 17th Conference on USENIX Security Symposium*, SEC '08, 2008.

[54] Jonathan Heusser and Pasquale Malacaria. Quantifying Information Leaks in Software. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, 2010.

[55] Johannes Hoffmann, Martin Ussath, Thorsten Holz, and Michael Spreitzenbarth. Slicing Droids: Program Slicing for Smali Code. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, 2013.

[56] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David 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, 2011.

[57] Jinseong Jeon, Kristopher K Micinski, and Jeffrey S Foster. SymDroid: Symbolic Execution for Dalvik Bytecode. *Technical Report CS-TR-5022, Univ. of Maryland*, 2012.

[58] David Kantola, Erika Chin, Warren He, and David Wagner. Reducing Attack Surfaces for Intra-Application Communication in Android. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, 2012.

[59] Kristen Kennedy, Eric Gustafson, and Hao Chen. Quantifying the Effects of Removing Permissions from Android Applications. In *Workshop on Mobile Security Technologies (MoST)*, 2013.

[60] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, and Junbum Shin. ScanDal: Static Analyzer for Detecting Privacy Leaks in Android Applications. In *Workshop on Mobile Security Technologies (MoST)*, 2012.

[61] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. SeL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, 2009.

[62] Kari Kostiainen, Jan-Erik Ekberg, N. Asokan, and Aarne Rantala. On-board Credentials with Open Provisioning. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS '09, 2009.

[63] Butler Lampson. Privacy and Security Usable Security: How to Get It. *Communications of the ACM*, 52(11), November 2009.

[64] Matthias Lange, Steffen Liebergeld, Adam Lackorzynski, Alexander Warg, and Michael Peter. L4Android: a Generic Operating System Framework for Secure Smartphones. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, 2011.

[65] Zhenkai Liang and R. Sekar. Etrace: An Extensible System Call Interposition Framework. `http://www.seclab.cs.sunysb.edu/etrace`.

[66] Zhenkai Liang, Weiqing Sun, R.Sekar, and V.N. Venkatakrishnan. Alcatraz: An Isolated Environment for Experimenting with Untrusted Software. *ACM Transactions on Information and System Security (TISSEC)*, 12(3), January 2009.

[67] Zhenkai Liang, V. N. Venkatakrishnan, and R. Sekar. Isolated Program Execution: An Application Transparent Approach for Executing Untrusted Programs. In *Proceedings of the 19th Annual Computer Security Applications Conference*, ACSAC '03, 2003.

[68] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural Support for Copy and Tamper Resistant Software. *ACM SIGPLAN Notices*, 2000.

[69] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, 2012.

[70] Petros Maniatis, Devdatta Akhawe, Kevin Fall, Elaine Shi, Stephen McCamant, and Dawn Song. Do You Know Where Your Data Are?: Secure Data Capsules for Deployable Data Protection. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS '11, 2011.

[71] Stephen McCamant and Michael D. Ernst. Quantitative Information Flow as Network Flow Capacity. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, 2008.

[72] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, 2008.

[73] Kristopher Micinski, Philip Phelps, and Jeffrey S Foster. An Empirical Study of Location Truncation on Android. In *Workshop on Mobile Security Technologies (MoST)*, 2013.

[74] Mohammad Nauman, Sohail Khan, and Xinwen 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, 2010.

[75] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically Rich Application-Centric Security in Android. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, 2009.

[76] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David 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.

[77] Marcus Peinado, Yuqun Chen, Paul Engl, and John Manferdelli. NGSCB: A Trusted Open System. In *Proceedings of 9th Australasian Conference on Information Security and Privacy*, ACISP '04, 2004.

[78] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code

Loading in Android Applications. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium*, NDSS '14, 2014.

[79] Raluca Ada Popa, Jacob R. Lorch, David Molnar, Helen J. Wang, and Li Zhuang. Enabling Security in Cloud Storage SLAs with CloudProof. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, ATC '11, 2011.

[80] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, 2011.

[81] Niels Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th Conference on USENIX Security Symposium*, SEC '03, 2003.

[82] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium*, NDSS '14, 2014.

[83] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. DroidChameleon: Evaluating Android Anti-malware Against Transformation Attacks. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIACCS '13, 2013.

[84] Ali Razeen, Bi Wu, and Sai Cheemalapati. SpanDex: Secure Password Tracking for Android. In *Proceedings of the 23rd Conference on USENIX Security Symposium*, SEC '14, 2014.

[85] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Trusted Language Runtime (TLR): Enabling Trusted Applications on Smartphones. In *Proceedings*

of the 12th Workshop on Mobile Computing Systems and Applications, HotMobile '11, 2011.

[86] Nuno Santos, Rodrigo Rodrigues, Krishna P. Gummadi, and Stefan Saroiu. Policy-sealed Data: A New Abstraction for Building Trusted Cloud Services. In *Proceedings of the 21st Conference on USENIX Security Symposium*, SEC '12, 2012.

[87] Dragos Sbîrlea, Michael G Burke, Salvatore Guarnieri, Marco Pistoia, and Vivek Sarkar. Automatic Detection of Inter-application Permission Leaks in Android Applications. *Technical Report TR13-02, Rice University*, 2013.

[88] Roman Schlegel, Kehuan Zhang, Xiaoyong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, NDSS '11, 2011.

[89] Daniel Schreckling, Joachim Posegga, and Daniel Hausknecht. Constroid: Data-centric Access Control for Android. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, 2012.

[90] Daniel Schreckling, Joachim Posegga, Johannes Köstler, and Matthias Schaff. Kynoid: Real-time Enforcement of Fine-grained, User-defined, and Data-centric Security Policies for Android. In *Proceedings of the 6th Workshop in Information Security Theory and Practice*, WISTP'12, 2012.

[91] Marcel Selhorst, Christian Stüble, Florian Feldmann, and Utz Gnaida. Towards a Trusted Mobile Desktop. In *Proceedings of the 3rd International Conference on Trust and Trustworthy Computing*, TRUST'10, 2010.

[92] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. AdSplit: Separating Smartphone Advertising from Applications. In *Proceedings of the 21st Conference on USENIX Security Symposium*, SEC '12, 2012.

[93] Weiqing Sun, Zhenkai Liang, R. Sekar, and V. N. Venkatakrishnan. One-way Isolation: An Effective Approach for Realizing Safe Execution Environments. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, NDSS '05, 2005.

[94] Richard Ta-Min, Lionel Litty, and David Lie. Splitting interfaces: Making Trust between Applications and Operating Systems Configurable. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, 2006.

[95] Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda. CleanOS: Limiting Mobile Data Exposure with Idle Eviction. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI '12, 2012.

[96] Amit Vasudevan, Emmanuel Owusu, Zongwei Zhou, James Newsome, and Jonathan M. McCune. Trustworthy Execution on Mobile Devices: What Security Properties Can My Mobile Platform Give Me? In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, TRUST '12, 2012.

[97] Amit Vasudevan, Bryan Parno, Ning Qu, Virgil D. Gligor, and Adrian Perrig. Lockdown: Towards a Safe and Practical Architecture for Security Applications on Commodity Platforms. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, TRUST'12, 2012.

[98] Carsten Weinhold and Hermann Härtig. VPFS: Building a Virtual Private File System With a Small Trusted Computing Base. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, 2008.

[99] Johannes 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, 2008.

[100] Chiachih Wu, Yajin Zhou, Kunal Patel, Zhenkai Liang, and Xuxian Jiang. AirBag: Boosting Smartphone Resistance to Malware Infection. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, NDSS '14, 2014.

[101] Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. The Impact of Vendor Customizations on Android Security. In *Proceedings of the 2013 ACM Conference on Computer and Communications Security*, CCS '13, 2013.

[102] Nan Xu, Fan Zhang, Yisha Luo, Weijia Jia, Dong Xuan, and Jin Teng. Stealthy Video Capturer: a New Video-based Spyware in 3G Smartphones. In *Proceedings of the Second ACM Conference on Wireless Network Security*, WiSec '09, 2009.

[103] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *Proceedings of the 21st Conference on USENIX Security Symposium*, SEC '12, 2012.

[104] Wei Xu, Fangfang Zhang, and Sencun Zhu. Permlyzer: Analyzing Permission Usage in Android Applications. In *Proceedings of the 24th IEEE International Symposium on Software Reliability Engineering*, ISSRE '13, 2013.

[105] Lok Kwong Yan and Heng Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proceedings of the 21st Conference on USENIX Security Symposium*, SEC '12, 2012.

[106] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and Xiaoyang Sean Wang. AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection. In *Proceedings of the 2013 ACM Conference on Computer and Communications Security*, CCS '13, 2013.

[107] Yang Yu, Fanglu Guo, Susanta Nanda, Lap-chung Lam, and Tzi-cker Chiueh. A Feather-weight Virtual Machine for Windows Applications. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, VEE '06, 2006.

[108] Xinwen Zhang, Onur Acıiçmez, and Jean-Pierre Seifert. A Trusted Mobile Phone Reference Architecture via Secure Kernel. In *Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing*, STC '07, 2007.

[109] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, Xiaoyang Sean Wang, and Binyu Zang. Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis. In *Proceedings of the 2013 ACM Conference on Computer and Communications Security*, CCS '13, 2013.

[110] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, CODASPY '12, 2012.

[111] Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, 2012.

[112] Yajin Zhou and Xuxian Jiang. Detecting Passive Content Leaks and Pollution in Android Applications. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium*, NDSS '13, 2013.

[113] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W. Freeh. Taming Information-Stealing Smartphone Applications (on Android). In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing*, TRUST '11, 2011.

[114] Zongwei Zhou, Virgil D. Gligor, James Newsome, and Jonathan M. McCune. Building Verifiable Trusted Path on Commodity x86 Computers. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, 2012.