The Dissertation Committee for Yuanzhong Xu
certifies that this is the approved version of the following dissertation:

Platform-level Protection for Interacting Mobile Apps

Committee:

Emmett Witchel, Supervisor

Lorenzo Alvisi

Roxana Geambasu

Keshav Pingali

Vitaly Shmatikov

Platform-level Protection for Interacting Mobile Apps

by

Yuanzhong Xu, B.E.; M.S.Comp.Sci.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin
August 2016

To everyone

Acknowledgments

I would like to thank Emmett Witchel for encouraging me to explore and innovate in the area of computer systems, and teaching me the value of practical research. I am also grateful to other great mentors, Weidong Cui, Marcus Peinado, and Vitaly Shmatikov, who have helped me with several research projects during the past few years. I also want to thank former members of our research group, Owen Hofmann, Alan Dunn, Michael Lee, and Sangman Kim, for their help during my early PhD career.

My five-year graduate school life has been fun and inspiring because of the help and support from many other friends and fellow graduate students: Sebastian Angel, Xue Chen, Natacha Crooks, Yu Feng, Martin Georgiev, Trinabh Gupta, Yige Hu, Jianyu Huang, Tyler Hunt, Youngjin Kwon, Zuocheng Ren, Chunzhi Su, Xinyu Wang, Yuepeng Wang, Chao Xie, Hongkun Yang, Sangki Yun, and Zhiting Zhu.

Finally, I want to thank my family for their encouragement that has been keeping me optimistic and patient.

Yuanzhong Xu

The University of Texas at Austin
August, 2016

Platform-level Protection for Interacting Mobile Apps

Yuanzhong Xu, Ph.D.
The University of Texas at Austin, 2016

Supervisor: Emmett Witchel

In a modern mobile platform, apps are mutually distrustful, but they share the same device and frequently interact with each other. This dissertation shows how existing platforms, like Android and iOS, often fail to support important data protection scenarios, and describes two systems to improve platform-level security.

First, many data leaks in existing platforms are due to the lack of information flow control for inter-app data exchanges. For example, a document viewer that opens an attachment from an email client often further discloses the attachment to other apps or to the network. To prevent such leaks, we need strict information flow confinement, but a challenge to enforce such confinement in existing platforms is the potential disruptions to confined apps. We present Maxoid, a system that uses context-aware custom views of apps' storage state to make information flow enforcement backward compatible.

Second, apps' abstraction of data has diverged from platforms' abstraction of data. Modern mobile apps heavily rely on structured data, and relational databases have become the hub for apps' internal data management.

However, in existing platforms, protection mechanisms are coarse-grained and have no visibility to the structures of apps' data. In these platforms, access control is a mixture of coarse-grained mechanisms and many ad hoc user-level checks, making data protection unprincipled and error-prone. We present Earp, a new mobile platform that combines simple object-level permissions and capability relationships among objects to naturally protect structured data for mobile apps. It achieves a uniform abstraction for storing, sharing and efficiently protecting structured data, for both storage and inter-app services.

Table of Contents

Chapter 1

Introduction

In modern mobile platforms, mutually distrustful apps from many different developers run on the same device. The platform is responsible for protecting apps from each other, treating them as different security principals. Despite the lack of mutual trust, apps communicate and exchange data with each other as much as they do with the platform. Individual apps are designed for specific functions, but many common tasks require functionality from multiple apps. For example, having an email app open an attached file using a document viewer app requires the two apps to cooperate. Moreover, many popular apps now provide standard services for third-party apps, such as storage provided by Google Drive and user authentication provided by Facebook. Storage and authentication have traditionally been the responsibility of the platform, but as a result of this shift, popular service apps become essential for supporting other apps that cannot be fully trusted.

Cross-app interactions are unavoidable because some apps provide essential functions. In addition to enforcing basic isolation between apps, the platform must mediate their interactions and enforce security policies. However, with standard security mechanisms in existing platforms, enforcing desired policies on cross-app interactions is often impractical or even impossible, because they either create burdens to developers and users or cannot be achieved at all. Existing platforms implicitly rely on developers and users to protect their data and determine what apps can be trusted, although in reality apps are mutually distrustful. To demonstrate the problem, we have identified two security challenges created by cross-app interactions.

First, inter-app data exchange increases the possibility of data leaks. In the email attachment example, even though the email app does not intentionally disclose the attachment to unauthorized parties, the chosen document viewer might not keep it private. Our study shows that popular document viewers often copy files to public storage which is shared by all apps; consequently, any app may subsequently read and send the file's contents to remote untrusted parties over the network. The platform, at best, can let the user decide whether to use the document viewer. Unfortunately, the user is often unaware of data leaks because they are more difficult to observe than functionality bugs; even if the user is aware of the risks, he or she may still use the app in order to open a specific type of file, choosing functionality over security.

Second, apps use ad hoc data structures to represent high-level semantics when they share data with each other. App-level structured data often involve complex relationships among objects, e.g., a photo album which includes photos, some of which have textual tags. The diversity of representations and semantics forces existing platforms to enforce access control using coarse-grained approaches, ignoring app-level semantics. An app could implement its own fine-grained checks on its ad hoc data structures, but that is tedious and error-prone because of the complex inter-related objects; in reality, most developers simply use the platform's coarse-grained mechanism. As a result, apps often get more access rights than they need, which is a violation of the principle of least privilege [Sal74]. For example, an image filter app is often given the permission for the entire photo gallery, even though the user only wants to process a small subset of photos. With such coarse-grained access control, if this app is malicious or contains security bugs, the entire photo gallery could be subject to data leakage or damage.

Existing platforms fail to adequately address the above challenges for cross-app interactions, and often leave data security up to individual mutually distrustful apps, resulting in serious problems [FWM+11,EOMC11,WXWC13, GJS14]. This dissertation presents two systems that improve the security mechanisms in mobile platforms with respect to these challenges, giving users and developers platform-level assurance for data security.

1.1 Information flow control

The desired security policies in many cross-app interaction scenarios involve information-flow properties, but these platforms contain insufficient mechanism to enforce a desired policy. For the email attachment scenario, the user would like the attachment not to be disclosed to unauthorized parties at any time, regardless of what apps open the attachment. However, in Android or iOS, once the document viewer is granted access to the attachment file, there is no restriction on what it can do with this file; it can copy the file to public storage, send it to other apps or upload it to a remote server. Without the ability to follow the flow of information, existing platforms pit security and functionality against each other, imposing a dilemma on the user: choose highly functional apps or choose better data security.

Maxoid [XW15] is our attempt to overcome the above limitation. Its security mechanisms track the execution context of an app instance and enforce information flow control. For example, when the document viewer receives a sensitive email attachment, it enters a confined mode where Maxoid guarantees it cannot further leak information about the attachment to unconfined apps or to the network. A challenge is that naïvely enforcing strong confinement can cause serious disruptions to existing apps, making the system unusable. Maxoid addresses this challenge by using a technique we call *custom views of state*, which creates multiple versions of data when necessary to make the confinement transparent. Maxoid has strong secrecy and integrity guarantees for both the app that shares data (e.g., the email client) and the app that receives data (e.g., the document viewer), without introducing complex programming models.

1.2 Abstraction for structured data

Apps' abstraction of data has diverged from platforms' abstraction of data. Internally, mobile apps heavily rely on structured data managed by relational databases, such as SQLite in Android and the SQLite-based library—Core Data—in iOS. However, the platforms' security mechanisms are coarse-grained and have no visibility to these structures; today, access control in mobile platforms is a mixture of basic coarse-grained mechanisms based on the traditional byte-stream abstraction inherited from UNIX (used for files, pipes, etc.), and ad hoc user-level checks spread throughout different system utilities and inter-app services. Each app presents an ad hoc API with ad hoc access-control semantics, different from those presented by the platform or other apps. This leaves apps without a clear and consistent model for managing and protecting access to users' data and leads to serious security and privacy vulnerabilities.

Earp [XHK$^+$16] is a system we propose to make platform-level security mechanisms support apps' abstraction of data. It exposes the relational model as platform-level uniform APIs for both storage and inter-app services, with structure-aware data protection. Being structure-aware means not only supporting fine-grained object-level access control, but also faithfully capturing the relationships among objects. Earp introduces *capability relationships* which are implied in the app's data model; for instance, having access to a photo album may transitively confer access to all photos contained in it. Capability relationships also make permission management feasible and efficient when access control is fine-grained.

Earp's unifying data-access abstraction is a *subset descriptor*. Subset descriptors are capability-like handles that enable the holder to operate on some rows and columns of a database, subject to restrictions defined by the data owner. Subset descriptors can also be downgraded and transferred to other apps. Moreover, capability relationships require transitively computing access rights which can be expensive, but by keeping computed access rights in subset descriptors that are created at run time, Earp avoids recomputing them on each operation, thus making both querying and access control efficient.

Chapter 2

Background

Like traditional operating systems such as UNIX, mobile platforms manage hardware resources, and provide common services allowing different apps to share the device. In fact, these platforms do use UNIX-like kernels—Android uses a customized Linux kernel, and iOS uses XNU, the kernel which is also used by Darwin.

The basic security model in traditional desktop OSes is motivated by the problem that different users may share the same machine. In this model, users are the security principals, and the kernel has a reference monitor that allows users to keep some of their files private while share some other files with other users.

In contrast, a mobile device typically has only one user, but he or she installs apps from many different companies and individual developers. There is no guarantee that these apps are trustworthy in terms of protecting the user's data. In reality, buggy, curious or even malicious apps have been causing serious security problems [HHJ+11, ZJ13]. As the principle of least privilege [Sal74] suggests, the platform needs to limit the access to user data and system resources for different apps, in order to reduce the amount of potential harm done by these apps. As a result, apps become the security principals, and they carry different access rights to data and system resources.

Mobile platforms provide rich APIs, which include not only the interfaces for physical device features such as sensors and cameras, but also standard ways to manage data with certain high-level semantics, such as photos and contact information. These semantics are typically considered as application-level concepts in desktop OSes, but in mobile platforms they are provided as standard platform APIs.

This chapter discusses software architecture, inter-app communication and security mechanisms in existing mobile platforms.

2.1 Software architecture

Figure 2.1 schematically shows the software architectures of several existing mobile platforms, with a comparison to desktop operating systems like UNIX or Linux. From an application's point of view, UNIX-like OSes provide a simple abstraction. Storage and communication share a unified byte-stream data model, with files and IPC channels such as pipes and sockets accessed via file descriptors [RT74]. The file descriptor API is implemented in the kernel, programs access storage and communication via system calls. The in-kernel reference monitor enforces access control at the granularity of an entire file or IPC channel.

Native mobile platforms. Native platforms are those where an app can be compiled into native code that directly runs on the device, without the

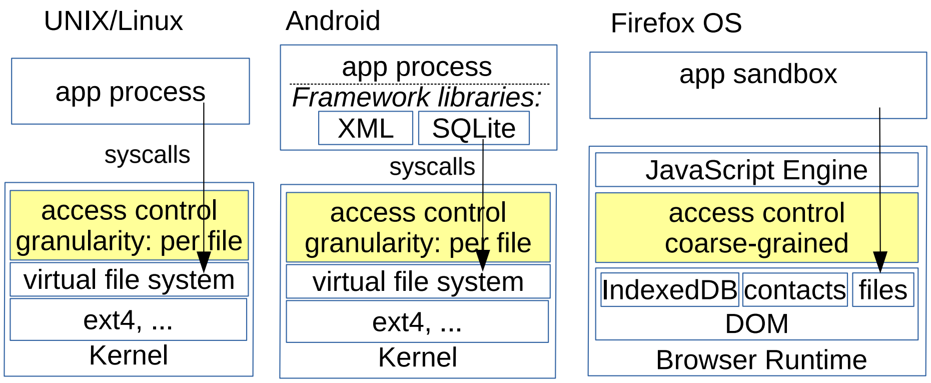


Figure 2.1: Comparison of software architectures in UNIX/Linux, Android, and Firefox OS.

requirement of a runtime. Although Android apps are typically written in Java and run in the Dalvik VM (a customized Java virtual machine), they may also include parts written in native-code languages like C and C++[1], which can directly make system calls to the kernel. Threfore, Android does not rely on the Dalvik VM to enforce security policies.

Besides raw files, Android has other standard data storage APIs such as databases and key-value stores. However, they are implemented as app-level libraries such as SQLite and XML. In Android, an app runs in its own process and has control of its entire user-level address space, so access control checks for storage need to be done in the kernel. From the kernel's point of view, databases and key-value stores are just files containing unstructured bytes. Consequently, the in-kernel reference monitor has no visibility into the internal structures of apps' data and cannot provide row- or column-level access control

[1] https://developer.android.com/ndk/index.html

or consider inter-object relationships when making access-control decisions. Similarly, iOS provides Core Data[2], a set of structured APIs for apps to manage and persist their data, but the structure of the data is not visible to the privileged reference monitor.

Browser-based mobile platforms. Several modern platforms support mobile apps purely written in Web code such as HTML5 and JavaScript, including Firefox OS, Google Chrome, WebOS, Tizen, Windows Runtime Apps and Ubuntu Touch. Their platform-level APIs expose database storage and high-level resource abstractions such as “calender”, “contacts” and “photo gallery” to apps; the implementation of these APIs is typically based on a customized, UI-less browser runtime. The browser runtime enforces access control checks for apps, since it cannot be bypassed by Web apps. Such enforcement is usually coarse-grained, e.g., at the granularity of the entire collection of contacts. From the app’s viewpoint, a call to a platform API is more like a system call than a traditional library call. The platform thus acts like the OS.

2.2 Inter-app communication

Modern mobile apps cooperate with each other and form an app ecosystem. To facilitate this trend, mobile platforms provide ways for apps to communicate.

[2] `https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/CoreData`

Foreground communication and invocation. Existing platforms allows a running app to invoke another app, with data attached to the invocation. This typically brings the invoked app to foreground. In Android, the invocation mechanism is called *intents*; an intent describes an invocation, which is first passed to Android's Activity Manager Service and then routed to a suitable receiving app. In iOS, invoking an app can be done by sending a *URL* that has a scheme registered by the app. In Firefox OS, a similar mechanism is called *MozActivity*.

Sharing via background services. Android supports inter-app data sharing via *content providers*[3]. A content provider is an app component that implements a background service that allows apps to read and write data. Android defines a standard, database-like API for all content providers, with operations including query, insert, update, and delete. Data in a content provider is identified by a *content URI* with optional query parameters. Despite the common API, the backend of a content provider can be implemented in arbitrary ways, e.g., using files, databases, or in-memory data structures.

Firefox OS allows an app to define a background service with *inter-app message ports*, which can send and receive JavaScript objects. Unlike Android, it does not define standard high-level APIs for such services.

In comparison, inter-app sharing in iOS is much more limited. Two

[3] `https://developer.android.com/guide/topics/providers/content-provider-basics.html`

independent apps cannot directly share data via the file system or background services. Instead, sharing often requires foreground operations and user involvement (e.g., choosing a file and invoking another app to open it), even with the *app extention* mechanism that allows a small set of UI elements to interact with other apps. Sharing may also be accomplished via copying to a system-wide shared API like the photo gallery.

2.3 Basic security model and enforcement

The security model in mobile platforms are app-centric. Apps are considered to be different principals, and the platform enforces access control in terms of what resources an app can access.

Private and public state. A high-level way to view data stored in a mobile platform is to differentiate *private state* owned by individual apps from *public state* shared among apps.

Private state is the data accessible to only the owning app. An app has full control over its private state, and can access it without explicit permissions. Note that although such state is private, nothing prevents the app from intentionally sharing its data to another receiving app if this is desired, e.g., sending via an invocation.

Allowing apps to have private state is essential to support the app-centric security model. An app may be entrusted by the user only for a specific task, and the user may want to keep the data related to the task private to that app; from the developer's point of view, it is also desired to protect the app's data from being accessed by untrusted apps or competing companies. For example, the Gmail app is expected to manage the user's emails and access Google account information, but not to access data owned by the Facebook app. The two apps can simply keep their sensitive data private.

In native platforms like Android and iOS, an app's private state is usually a dedicated file directory, and the kernel ensures that it is by default only accessible to the owning app—Android assigns unique UIDs to apps and uses file permissions in Linux, while iOS uses a sandbox supported by its kernel that can confine an app's access to the file system. Structured APIs like databases are provided as libraries, and they use file-backed storage; the platform considers them as raw files when enforcing access control. In Browser-based platforms, private state includes some HTML5 storage APIs such as IndexedDB, and access control is enforced by the browser runtime.

Public state includes standard storage APIs available to all apps, such as file storage on an SD card, the user's contact list and photo gallery. Access to such resources typically requires platform-defined permissions for the app. Enforcement of access control is usually specific to platforms and the implementation of these shared resources.

In fact, some shared resources are implemented in *built-in apps*, e.g., the photo gallery in Android is part of the Media Provider app. Depending on the context, we can either think that the gallery app is *part of the platform*, or that it is *an app that shares data to other apps*. Chapter 3 chooses the former interpretation while Chapter 4 chooses the latter, because they focus on problems from different perspectives.

Permissions. Mobile platforms associate system resources with *permissions*. An app carries a list of permissions that are granted either upon installation or at run time as approved by the user. For example, an app requires corresponding permissions in order to access cameras, sensors, SD card storage, photos, and contacts. Despite the significant difference among these resources, the simple permission mechanism works for them in the same coarse-grained manner—it simply accepts or denies any request to the resource by checking the associated permissions without interpreting any high-level semantics.

In iOS, apps need to request permissions at run time, by showing prompts to the user. In earlier versions of Android, permissions can only be granted at install time. That means an app specifies a list of permissions in its package manifest, and the user has to accept all of them in order to install it. This design was aimed at improving user experience by avoiding permission prompts. However, beginning in Android 6.0, apps can request critical permissions at run time, at the granularity of permission groups[4], and the user is allowed to revoke them. On the other hand, iOS historically requires run-time permission granting.

Some platforms also allow an app to define its own permissions when sharing its data to other apps. For example, an Android app can define a

[4] `https://developer.android.com/training/permissions/requesting.html`

content provider to serve data for other apps, but may require the other apps to obtain a permission it defines.

Access control in Android content providers. Android's content provider mechanism is relatively complex, and deserves detailed discussion here. An app can statically define permissions required for other apps to access its content provider, specified in its manifest—an XML file that comes with the app's distribution package. It can define read/write permissions for the entire content provider, or separate permissions for different subsets identified by different URI paths. However, even with path-level permissions, such permissions are still limited to coarse-grained categories (e.g., Media Provider could use different path-level permissions for images, audio and video, although in fact it only uses a single permission for them all) because it is impossible to assign different permissions to dynamically created objects, nor specify custom policies for different client apps.

Android also has another mechanism[5] for fine-grained, temporary access granting. However, although Android calls it "URI permissions", it is not a way to specify or enforce security policies; instead, it can be considered a run-time capability-passing mechanism, similar to passing a file descriptor to another process in Linux. Android mostly uses them to involve the user in access-control decisions; for example, when the user clicks on a document and

[5] `https://developer.android.com/guide/topics/security/permissions.html#uri`

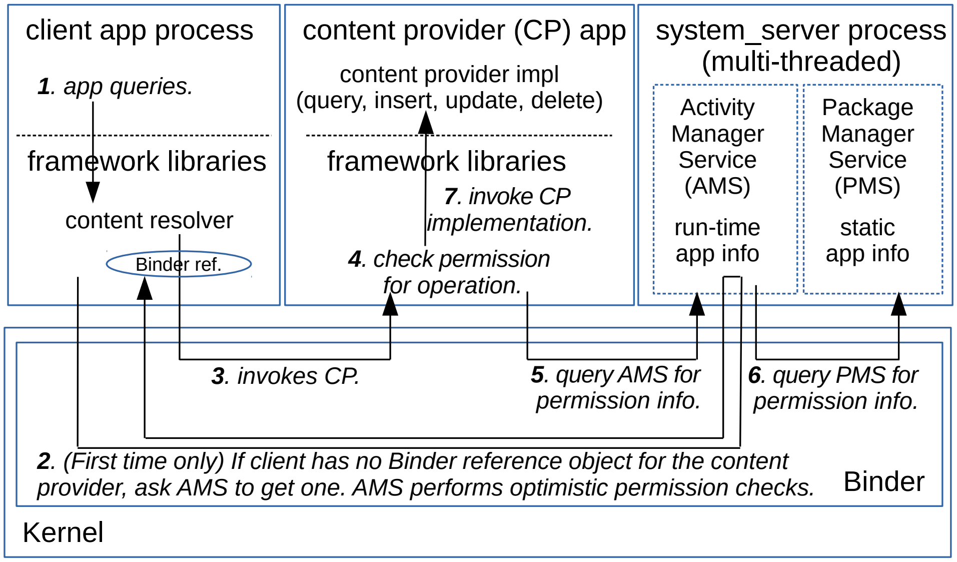

client app process
1. app queries.
framework libraries
content resolver
Binder ref.
content provider (CP) app
content provider impl
(query, insert, update, delete)
framework libraries
7. invoke CP implementation.
4. check permission for operation.
system_server process (multi-threaded)
Activity Manager Service (AMS)
run-time app info
Package Manager Service (PMS)
static app info
3. invokes CP.
5. query AMS for permission info.
6. query PMS for permission info.
2. (First time only) If client has no Binder reference object for the content provider, ask AMS to get one. AMS performs optimistic permission checks.
Binder
Kernel

Figure 2.2: Android apps communicate using a content provider.

chooses an app to open it, a URI permission for this document is dynamically passed to the receiving app.

In Android, low-level inter-process communication (IPC) is based on *Binder*, a kernel-level mechanism that implements a remote procedure call (RPC) framework for different app processes or threads to invoke each other. Android's customized Linux kernel contains a driver for Binder. In a Binder connection, the client process holds a Binder reference object. Reference objects cannot be constructed by the client itself, but they can be passed to other processes like file descriptors. Binder is used by the content provider framework.

All running apps have been initialized with Binder connections to core system services, such as Activity Manager Service and Package Manager Service, which typically run in a multi-threaded process named `systerm_server`.

Figure 2.2 illustrates how a client app communicates with a content provider and how access control is enforced. The processes of these two apps directly communicate via the Binder mechanism, on top of which Android's framework libraries implement the content provider interface. The client-side interface is called a *content resolver*, which translates the app's queries to low-level Binder calls.

The content resolver first checks if a connection has been established with the content provider. If not, it invokes Activity Manager Service to get a Binder reference object for the content provider. Activity Manager Service performs preliminary, optimistic permission checks—if there is any chance that the client may be permitted to access the content provider regardless of what operation it may use, Activity Manager Service passes a reference object to it. If the connection has been established, the content provider is invoked using the connection.

On the service side, when a Binder request is received, the content provider's library code first retrieves low-level credentials (e.g., UID and PID) about the calling app process as provided by the kernel, then invokes Activity Manager Service to query whether the requesting process has the required permission to perform the requested operation. Activity Manager Service has run-time information about all app processes, and it may also communicate with Package Manager Service to get static information about different apps' permissions. If the request passes the permission check, the library code calls the app-specific implementation of the content provider, then returns result.

We summarize the security mechanisms for content providers.

- Permissions are static and coarse-grained. For example, Android's built-in Contacts Provider defines two permissions in its manifest, for reading and writing the entire contact database, respectively.

- Run-time capability passing can be fine-grained, but the decision whether to pass a capability is up to the app. For example, an app that has access to Contacts Provider can pass the capability for a specific contact to another app, but it needs to decide whether this should happen, e.g., whether this is permitted by the user.

- Access checks are performed by framework library code that runs in the same process as the content provider app, with reliable request information provided by the kernel and system services (Figure 2.2).

Chapter 3

Maxoid: transparently information flow confinement

For mobile apps, the tension between the diversity of providers and the goals of a seamless mobile experience creates a security problem. Apps from different developers must work together, but they have no reason to trust each other. Mobile platforms like Android and iOS provide app-centric security models where apps are treated as different principals, to protect each app's private data, and control each app's access to shared data by specifying a variety of permissions (Section 2.3). However, such an intuitive model is not sufficient to protect confidentiality or integrity for common scenarios where the user would like two or more apps to cooperate on sensitive data.

For example, email apps must invoke external programs to view attachments, cloud storage apps must invoke external editors, and a comparison shopping app may need a bar code reader app. In these examples, data exchange happens across apps. From the app-centric security perspective, it is desirable to keep the data private to the original owning app (email, cloud stor-

This chapter is based on previous publication [XW15] "Maxoid: Transparently Confining Mobile Applications with Custom Views of State", by Yuanzhong Xu and Emmett Witchel, in the 10th ACM European Conference on Computer Systems (EuroSys), Bordeaux, France, April 2015. My contributions to this publication include investigating data leakage problems in Android, designing the Maxoid state model, implementing the Android-based prototype, building use cases and evaluating performance overheads.

age, shopping); however, the owning app does not have sufficient functionality to process the data. We call the app that needs to invoke other helper apps the *initiator* and the invoked app the *delegate*, and we say that the delegate runs *on behalf of* the initiator. In existing platforms, once the initiator shares sensitive data with the delegate, it has no control on how the delegate uses the data. For instance, the delegate may copy the initiator's sensitive data to public storage (see Section 3.1.2).

The app-based security model in modern mobile systems allows a new balance of usability and security for initiator and delegate apps that is not available to desktop or server systems. Based on the clear distinction of apps' private and public (shared) state, it is possible to reason about security requirements for inter-app cooperation with fairly simple, coarse-grained information flow mechanisms that require little or no change to existing apps and without requiring new, complex policies.

We propose Maxoid[1], a new security model that provides secrecy and integrity for cooperating apps. We have built a prototype of Maxoid by modifying Android 4.3.2. Maxoid allows delegates to access initiator private state, but prevents delegates from leaking these secrets to other apps or transferring them over the network; delegates may update initiator private state or public state to return results, but Maxoid allows the initiator to selectively commit or discard those updates to prevent unwanted modifications by dele-

[1] The name is a contraction of The Matrix and Android, because Maxoid composes a custom reality for delegates on Android.

gates. Conversely, Maxoid also protects delegates by disallowing the initiator from reading or writing its delegates' private state.

Maxoid achieves its security goals while minimizing disruption to delegates by presenting different *views* of private and public state to initiators and delegates. A delegate's view transparently confines its access to persistent state like files and data in system content providers (e.g., Media). Delegates can still access resources to which they have permission (except the loss of network connection when the confinement begins), without violating Maxoid's security properties. Controlling views of state, e.g., by using a union file system and a copy-on-write SQL proxy, transparently provides a coarse-grained mechanism to control information flow.

Maxoid prioritizes backward compatibility and ease of adoption. It is fully compatible with legacy Android apps when the new Maxoid features are not used for them. Even when being used to confine delegates, Maxoid can be completely transparent, i.e., it can support unmodified delegate apps with full security guarantees. It also provides simple (often optional) APIs for developers to improve usability. For example, some of a delegate's data may be cleared by Maxoid for transparency by default, but it may alternatively use Maxoid APIs to keep persistent state, like a list of recently accessed files. However, this state is only accessible when the delegate is run by that same initiator. Thus, a PDF viewer that runs on behalf of an email client can have previous email attachments in its recently opened list, but these attachments will not be visible when the PDF viewer does not run on behalf of the email client.

Finally, Maxoid has negligible overhead for initiators compared to unmodified Android; for delegates, it adds a small overhead for most operations though it slows down certain worst-case microbenchmarks.

3.1 Motivation and overview

This section describes the private and public state in Android, problems found in case studies, how a naïve information flow control solution suffers from poor usability, and an overview of Maxoid.

3.1.1 Private and public state in Android

In Android, each app is assigned a dedicated UID, which isolates apps from each other. An app's private state includes shared preferences,[2] internal file storage, and private SQLite databases. All of them are stored as private files of the owning app, with the interface to the key-value store and database provided by user-level libraries.

Public state includes external file storage (e.g., SD card) and system content providers (Downloads, Media, User Dictionary, Contacts, etc.). System content providers are implemented as built-in apps, but we treat them as part of the Android platform in this chapter, because they provide essential standard APIs for data storage (Section 2.3).

In earlier versions of Android, an app can either have no access to

[2] Though called shared preferences, it is actually a private key-value store, see `http://developer.android.com/guide/topics/data/data-storage.html`.

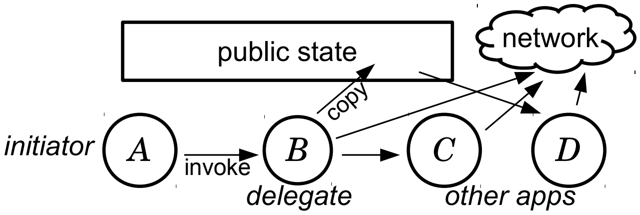


Figure 3.1: A delegate app (B) can cause leakage of data from an initiator app (A), by copying it to other apps and the public state, or sending over the network.

external storage, or have access to all files on it. Starting from Android 4.4, an app may have partial access to external storage; each app is granted access to a dedicated directory without explicitly asking for permission. However, apps with permission for external storage can still access all files on it. Therefore, we still consider the entire external storage as public state.

3.1.2 Case studies

We analyze the behavior of some popular Android apps that collaboratively execute while sharing sensitive data. We categorize these apps into two types: 1) data processing apps, which can be typically used as delegates, and 2) apps that need help from data processing apps, which can be used as initiators. One theme that emerges is that Android's access control model provides no information flow control on sensitive data, which limits how effectively it can enforce security protections. Figure 3.1 is an high-level illustration of how data leaks can happen in a typical initiator-delegate scenario.

Category	# of Apps	Sample App	Operation	State left after the operation	
				Private state	Public state
Document viewer, editor, converter	17	Adobe Reader	open a file via a URI	XML: recent files.	File copy on SD card.
		Kingsoft Office	open a file via a URI	ADF: recent files.	File copy on SD card. A thumbnail on SD card. Entries in a DB on SD card.
Scanner	20	Barcode Scanner	scan a QR code	DB: recent scans.	
		Cam Scanner	scan a file	DB: recent scans.	Image saved to SD card. Thumbnail on SD card. Log file on SD card.
Photo	30	Camera MX	take a photo		Photo saved to SD card. Entry in Media Provider.
			edit a photo		Entry in Media Provider.
Media	10	VPlayer	play a video	DB: playback history.	Thumbnail on SD card.

Table 3.1: State left after apps process their target data. In the private state column, XML indicates state saved in the shared preference key-value store; DB indicates state saved in an SQLite database; ADF indicates state saved in files with app-defined formats.

Data processing apps. We manually study 77 popular Android apps for processing different types of data, such as documents, media files, and QR codes. These apps are selected based on popularity and relevance from Google Play. We find that, after processing data, these apps leave traces of that data that can be accessed by other apps. Table 3.1 summarizes how different classes of apps leak state. Currently, there is no careful control of state at the application level, so Maxoid aims to provide it at the system level.

Apps that need help of others. We analyze four Android apps that need the help of other apps.

I. Dropbox. Dropbox hosts the user's files, but has very limited support for processing files. When the Dropbox app fetches a file from its server, it saves the file to a directory in public external storage to allow other apps to open it. Therefore, the Dropbox client does not provide privacy on its files. Whenever another app changes a file, Dropbox automatically syncs this change to its server, even if this change is unintended. This behavior provides no integrity for Dropbox's files.

II. Google Drive. Google Drive is similar to Dropbox, but 1) it caches downloaded files in its private internal storage; 2) it can save encrypted files to external storage for offline access, which will be decrypted and cached in internal storage when the user opens them. Google Drive makes internal cached files world-readable to allow other apps to open, but the path names include random strings and other apps cannot list entries in the parent directory. Thus invoked apps only know how to access specific files that Google Drive discloses to them via invocations. However, they can leak information about the files that have been disclosed to them. (see Table 3.1).

III. Email. Emails often contain attachments. By default, Android's built-in Email app saves an attachment file in its private internal storage for security. The user can explicitly save an attachment to external storage and its metadata to the Downloads provider. To allow another app to open the private internal file, Email uses Android's per-URI permissions: it defines a content provider that maps a content URI to an attachment file, then invokes the other app with the corresponding URI, and sets the flag `FLAG_GRANT_READ_-`

URI_PERMISSION. Now the invoked app can open this URI to get a ParcelFileDescriptor. The actual file is still opened by Email's process, but the file descriptor is passed to the invoked app. This mechanism only grants the invoked app one-time permission on the single file. However, the invoked app can still copy this file to its private state or public state (Table 3.1).

IV. Browsers. Chrome and Android's built-in Browser app support incognito mode to avoid leaving traces about the user's browsing history on the device. However, neither browser supports incognito download. In an incognito tab, a user-downloaded file will be saved to external storage and added to the Downloads provider, which maintains index and metadata for downloaded files. Even if the browsers were modified to store the files in private internal storage, and adopt a per-URI permission approach to allow other apps to open them, the same problems would still exist as with Email, since the browsers cannot erase data left by other apps. Even a browser with perfect incognito mode would not address the safety of input data. For example, if the user reads a URL from a QR code scanner app and opens it in a browser, the browser's incognito mode cannot erase the data's history in the scanning app.

In summary, the fundamental problem with app collaboration in Android is a lack of an information flow security mechanism that would allow another app to receive sensitive data, but then limit the receiving app's ability to communicate once it has read that data.

3.1.3 Information flow tracking and challenges

Additional information flow control mechanisms are needed to secure the use cases in Section 3.1.2. A potential solution is to perform taint tracking on apps' private data. The system allows one app to send its private data to another app, but the data is labeled as tainted. Then the receiver is confined such that any of its data depending on the received data will also be tainted, and disallowed from being written to public storage or the network. This approach is in line with previous decentralized information flow control (DIFC) systems.

Difficulty in programmability. Typical DIFC systems are not designed to be backward compatible with legacy applications. Applications need to be re-written to comply with the security rules in those systems. Understanding subtle data flows makes it difficult to adapt complex applications to fine-grained information flow tracking [EKV$^+$05].

However, our goal is to support legacy applications. Naïvely applying previous approaches would cause serious usability issues.

Uncontrolled taint propagation. Legacy apps often do not distinguish public input and private input from other apps. For example, when Adobe Reader opens a PDF file, it does not take extra care of controlling data propagation if the file is a private attachment from Email; in reality, it creates an entry in the list of recent files, and makes a copy of the attachment and stores it on the public SD card (Table 3.1).

To secure this use case, a taint tracking system would need to label the attachment as tainted, and control the propagation of data depending on it. It may disallow Adobe Reader from writing tainted data (such as a copy of the file) to the SD card or the network. However, such restrictions would probably break the normal operation of Adobe Reader, because it would get unexpected permission errors.

An alternative approach is to still allow Adobe Reader to write tainted data to the SD card, but to keep the taint on the written data. Writing tainted data to the network is still disallowed, because the platform cannot track taint propagation outside the device. This approach may not directly break Adobe Reader, but it suffers from the problem of uncontrolled taint propagation. The SD card is a public resource, which means if other apps read tainted data on it, they would be tainted as well. Different apps would collectively propagate taint throughout the device, making many apps unable to write to network.

Figure 3.2 summarizes the dilemma when using such naïve information flow tracking approaches.

Granularity of taint tracking. In general, a more fine-grained taint-tracking mechanism tends to suffer less from usability problems caused by false positives. However, fine-grained mechanisms also tend to have more complexity and performance overhead. TaintDroid [EGC$^+$10] is a fine-grained taint track-

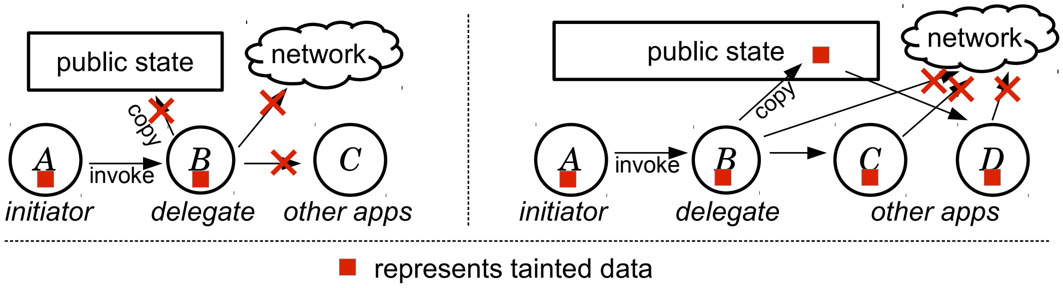


Figure 3.2: Dilemma of naïvely applying information flow tracking approaches. The approach shown on the left causes significant disruptions to the delegate by disabling access to public state and communication with other apps; the other approach shown on the right results in uncontrolled taint propagation that disables many apps from accessing the network.

ing system with moderate overhead on Android, but it does not track implicit data leakage via control flows.

3.1.4 Overview of Maxoid

To solve the above usability problems, Maxoid controls the propagation of tainted data by maintaining extra copies of data when necessary, and presenting transparent views of these data for confined apps to keep backward compatibility.

This technique allows Maxoid to adopt a fairly coarse-grained, conservative taint tracking mechanism while remaining usable. In Maxoid, once an app receives private data from another app, all of its outputs are considered tainted and thus protected by creating extra copies. The coarse-grained approach avoids much of the potential complexity and performance penalty in taint-tracking systems.

Definitions. Maxoid differentiates the execution context of an app instance running **on behalf of another**. In Maxoid, an app can run on behalf of itself, in which case it executes identically to how it would in Android. But if an app executes on behalf of another app, there are system facilities to manage information propagation.

App B's instance running on behalf of app A is denoted as B^A, where A is called the **initiator** app of B^A, and B^A is called a **delegate** of A.

Like in Android, an app can **declassify** its private data by writing it to public state, or sending it via IPC to other apps. Maxoid does not prevent A from mistakenly declassifying its own private state; it prevents B^A from leaking A's sensitive data via public writes or IPC.

Maxoid confines B^A so it can safely access A's private data. To make the confinement transparent to B^A, Maxoid creates custom **views of private and public state** for B^A. In these views, B^A can still access a resource as long as B normally has the permission (see Section 3.2.1).

Maxoid confinement is **invocation-transitive**. When B^A invokes another app, the invoked instance is forced to be a delegate of A, e.g., C^A (see Section 3.2.4).

Augmented delegate access right. Input to B^A is even more permissive than B's normal execution – B^A can also read A's private state. B^A can still observe other apps' updates to public resources after B^A starts. Moreover, B^A can still write to all allowed resources, and it will read its own writes, but these writes are transparently confined by Maxoid. B^A does not need to know it is executing on behalf of A, which allows Maxoid to support unmodified apps.

Network. In keeping with Maxoid's coarse-grained design philosophy, delegates are prevented from accessing the network, because Maxoid cannot control data flow in the network. Since network disruption is common in the mobile environment, cutting off network access is typically tolerated by apps. The delegate still has access to any data fetched from the network prior to its starting to run on behalf of an initiator. When the delegate is next run on behalf of itself (as an initiator), its access to the network is restored. Lack of network access for delegates means that Maxoid does not support scenarios where B^A needs to send A's private data to a server for processing (although A still has the option to invoke B to do that insecurely as in Android). We could avoid cutting off network access by extending Maxoid into apps' backend services, if they were all hosted on a trusted cloud, and preventing apps from accessing network resources other than the trusted cloud, like in πBox [LWG+13].

IPC. Maxoid tracks and controls inter-app communication to enforce its security properties. It also allows initiators to specify their security requirements using Android intents—Android's inter-app invocation mechanism.

3.1.5 Threat model

Maxoid protects initiators from arbitrary malicious delegates. The delegate apps can be written in Java and run in the Dalvik VM, or written in

C and compiled as native binaries. This is because Maxoid's security enforcement is implemented in trusted system services and the kernel. Delegates can directly access private data of their initiators, but Maxoid controls their output to avoid data leakage and unexpected modifications.

Maxoid also protects delegates from malicious initiators. Being an initiator does not mean the app is privileged; like in Android, it is still prevented from reading or writing private data of other apps, including its delegates.

Maxoid does not prevent an app from mishandling its own private data. It does not stop an initiator from mistakenly leaking its own private data, or mistakenly handling the interactions with their delegates which might compromise data integrity.

Maxoid assumes the operating system kernel and trusted system services are not compromised. Side channel attacks are out of our scope.

3.2 State model and Maxoid architecture

Maxoid presents different transparent views of private and public states to initiators and delegates. Some data in these views may have different versions; maintaining multiple versions of data is a key technique in Maxoid that resolves the problem of taint propagation. We introduce several notations for views of state.

- $Priv(x)$: the view of private state for app instance x.
- $Pub(x)$: the view of public state that Maxoid presents to x. Note that

this includes resources that x may not have permission.[3]

- $Pub(all)$: the data shared by all apps. If x is an initiator, $Pub(x) = Pub(all)$.

Whether an app runs as a delegate or an initiator, it can access everything in its view of private state, and everything in its view of public state for which it has the corresponding Android permissions (decided at install time).

The goal of Maxoid is to improve security for A by confining B^A in such a way as to minimize disruption and code changes to Android, A, and B. Maxoid achieves the following security goals and usability goals.

S1. Secrecy of the initiator. Only A and delegates of A can access A's private state. When B no longer runs on behalf of A, it cannot observe data depending on A's private state, unless A declassifies it, e.g., by writing it to public state, or sending it via IPC to other apps.

S2. Integrity of the initiator. When B^A updates A's private or public state, A has the ability to revert to the previous version. In fact, Maxoid requires A or the user to commit B^A's update to make it the default version for A and other apps not executing on behalf of A; otherwise, the update is only visible to A and A's delegates.

[3] x can actually access $Pub(x) \cap Perms(x)$, where $Perms(x)$ is the set of Android permissions that x has for public resources. For simplicity, we do not explicitly mention $Perms(x)$ in this section.

S3. Secrecy of the delegate. A cannot learn the private state of B^A unless B^A declassifies it.

S4. Integrity of the delegate. First, A cannot write to B^A's private state; second, when B no longer runs on behalf of any other app, Maxoid restores the private state as it was right before it was last started as a delegate. Having run on behalf of other apps does not modify B's private state.

In addition to the security guarantees, the design of Maxoid is guided by a principle that we call *minimum isolation*: whenever a data flow is safe, it should be allowed in order to minimize disruption. In addition to minimum isolation, Maxoid strives to be backward compatible. Minimum isolation guides ***U1*** and ***U2***, while backward compatibility guides ***U3***.

U1. Initial state availability. When B^A is started, $Pub(B^A)$ and $Priv(B^A)$ contain all data available in $Pub(all)$ and $Priv(B)$ up to that point. Maxoid does not create a blank initial environment for delegates, where a delegate would lose the user's normal preference settings and useful data collected previously.

U2. Update visibility. First, an initiator's update to public state can be observed by all app instances, including delegates of any initiator. Second, a delegate's update to public state (e.g., $Pub(B^A)$) should be observed by its initiator (e.g., A) and all delegates (including itself) of the same initiator (e.g.,

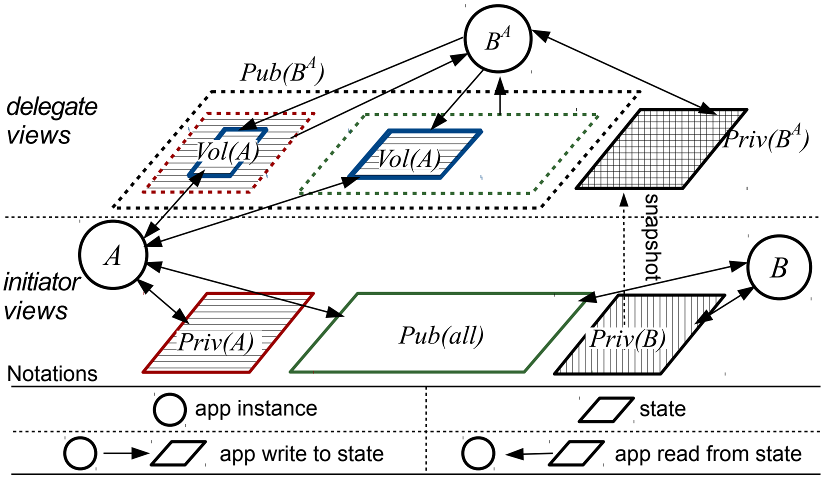


Figure 3.3: Overview of Maxoid confinement. Hatching in a state box indicates taints: $Priv(A)$ and $Priv(B)$ are the sources of taints, $Vol(A)$ is tainted by $Priv(A)$, and $Priv(B^A)$ is tainted by both $Priv(A)$ and $Priv(B)$.

C^A).

U3. Transparency to delegates. Maxoid should support unmodified delegates by maintaining the same API to access state as Android. B^A is always allowed to read/write $Priv(B^A)$; B^A is allowed to read/write a resource in $Pub(B^A)$ as long as B has the permission to read/write this resource in $Pub(all)$.

3.2.1 Confining delegates by custom views

Figure 3.3 illustrates how Maxoid confines a delegate. Solid arrows represent possible read/write by an app instance to a state. We describe the confinement and show how it achieves the security and usability goals.

Views. For initiators, the views of private and public state are identical to those in Android.

For a delegate B^A, $Priv(B^A)$ is initialized as a snapshot of $Priv(B)$ (***U1***), and any update by B^A is made copy-on-write. As a result, B^A's private writes are confined in $Priv(B^A)$ and can not affect $Priv(B)$ (***S4***).

Initially $Pub(B^A)$ consists of $Pub(all)$ (***U1***) and $Priv(A)$. By including $Priv(A)$ in B^A's view of public state, Maxoid naturally grants B^A the permission to access $Priv(A)$. However, all writes by B^A to $Pub(B^A)$ are redirected to the **volatile state** of A, or $Vol(A)$, such that B^A cannot directly overwrite $Pub(all)$ or $Priv(A)$ (***S2***).

All delegates of A share the same $Vol(A)$, and the same view of public state. We use $Pub(x^A)$ to denote the view for all delegates of A, where x is not a specific app. $Vol(A)$ is defined as the set of data written by all of A's delegates to $Pub(x^A)$. $Pub(x^A)$ is a transparent, merged view of $Pub(all) \cup Priv(A)$ and $Vol(A)$ (see Section 3.2.3).

Information flows. A directed path of solid arrows in Figure 3.3 represents an information flow. Maxoid doesn't use fine-grained taint tracking [EGC$^+$10], but enforces conservative rules to guarantee security.

1. $Priv(A) \rightarrow B^A \rightarrow Vol(A)$. This indicates that $Vol(A)$ may depend on $Priv(A)$, i.e., $Vol(A)$ is tainted by $Priv(A)$. Thus $Vol(A)$ is only visible to A and delegates of A (***S1***).

2. $Priv(A) \rightarrow B^A \rightarrow Priv(B^A)$. $Priv(B^A)$ is thus tainted by both $Priv(A)$ and $Priv(B)$ ($Priv(B^A)$ is initially forked from $Priv(B)$). Therefore, B^A is the only app instance that can access $Priv(B^A)$ (***S1***, ***S3***).

3. $Priv(B^A) \rightarrow B^A \rightarrow Vol(A)$, but $Vol(A)$ is not tainted by $Priv(B)$, because $Vol(A)$ is part of $Pub(B^A)$ and B^A already declassifies the writes to $Vol(A)$, i.e., removes the $Priv(B)$ taint; however, it has no power to remove the $Priv(A)$ taint on $Vol(A)$. Maxoid, like Android, considers every write by x to $Pub(x)$ a declassification.

4. $Vol(A) \leftrightarrow A$, A can observe and control its delegates' updates to $Pub(x^A)$ (***U2***).

5. A cannot read or write $Priv(B^A)$ (***S3***, ***S4***).

Transparency (*U3*). The security properties (***S1***- ***S4***) are automatically enforced by Maxoid presenting B^A custom views of state. B^A can still read/-write data in $Priv(B^A)$ and $Pub(B^A)$, without extra app logic to obey security rules.

3.2.2 Evolving views of private state

History of a delegate's private state. When B^A starts, $Priv(B^A)$ is forked from $Priv(B)$, as required by initial state availability (***U1***). When B no longer runs on behalf of A, its private state is resumed to the version that was forked. If B makes updates to $Priv(B)$, then $Priv(B^A)$ and $Priv(B)$ will diverge. The next time B^A runs, Maxoid cannot merge them.

In that case, if B is not aware of Maxoid, to maintain transparency, we could either 1) discard the old $Priv(B^A)$, and fork from $Priv(B)$ if it diverges from the old $Priv(B^A)$; or 2) keep using the old $Priv(B^A)$. Either way, some updates are invisible to B^A, although it is safe to let B^A see them. We choose the first option, for several reasons. First, the user can update his/her preferences while normally using B, and those updates will be in effect when he/she uses B as a delegate of any other app; second, B^A does not have network access but B could fetch data from the Internet, thus $Priv(B)$ may contain resources that B^A cannot obtain. Note that $Priv(B^A)$ will not be discarded when B is consecutively invoked as a delegate for any initiator.

Persistent private state. Nevertheless, if the delegate app is aware of Maxoid, it can use a Maxoid API to improve its usability. Maxoid splits a delegate's private state into two parts: 1) the **normal private state** as in Android, $nPriv(B^A)$, and 2) the **persistent private state**, $pPriv(B^A)$.

$nPriv(B^A)$ will be discarded if it diverges from $Priv(B)$, and will be reforked from it. $pPriv(B^A)$ will not be discarded (unless A explicitly requests so), and B^A can use it to store data that is persistent across invocations even if B updates $Priv(B)$ between invocations of B^A. For different initiators, delegates have different isolated views of persistent private state, e.g., $pPriv(B^A)$ and $pPriv(B^C)$ are isolated. Figure 3.4 demonstrates how $pPriv$ and $nPriv$ evolve over time.

$pPriv$ is a new API to delegates which is not transparent. However,

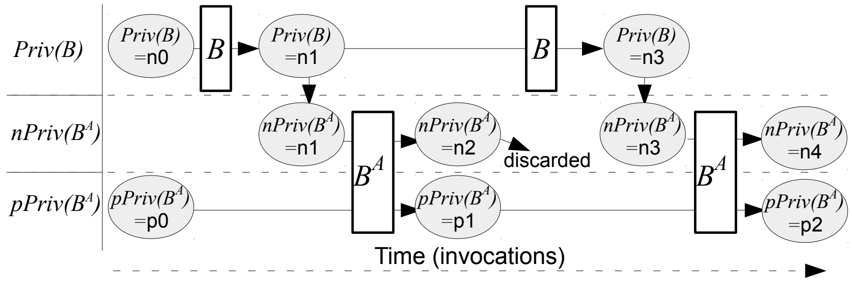


Figure 3.4: Normal and persistent private states evolving over time. A solid box is an app instance running for a period. An ellipse shows the value (version number) of a state before or after an invocation.

this API is *optional*, and exists only for improving usability. For instance, if a document viewer runs normally, it can store entries of recent files in a database that belongs to its normal private state. If it runs on behalf of another app, it can store the entries in a database that belongs to its persistent private state; other unimportant updates like cache files can still be stored in the normal private state. When it is started as a delegate, it can generate a list of recent files merged from both databases.

3.2.3 Public state and volatile state

In Android, a public resource can be located via a file name or a URI (for content providers), which we refer to as a **name**. The entire public state can be viewed as a set of name-value pairs.

Maxoid needs to create extra volatile copies of data when delegates write to their views of public state, to prevent $Pub(all)$ from being tainted. Maxoid does not take a full snapshot of the entire $Pub(all)$ when a delegate starts. Instead, it adopts a **unilateral per-name copy-on-write** mechanism.

If none of A's delegates has updated a public resource, the same copy of this resource is shared in both $Pub(x^A)$ and $Pub(all)$; B^A can see updates to this resource by initiators. Once a delegate of A updates a public resource, Maxoid creates a volatile copy of this resource for all delegates of A. From this point on, B^A only sees the volatile copy and cannot observe the updates from non-delegates, until A removes this volatile copy; however, this does not affect other resources.

This copy-on-write mechanism is unilateral, because it only happens for writes from delegates. With this mechanism, delegates of A may observe some resources updated themselves, but some other resources updated by initiators. If the two sets of resources have dependencies, consistency issues might occur. However, inconsistencies in public resources are common in Android because they are rarely protected by system-wide locks. At minimum, Maxoid guarantees that all of A's delegates can read their writes.

We do not use full snapshots of $Pub(all)$, for two reasons. First, creating a full snapshot for a delegate would make it unable to observe later updates from initiators to any resource in $Pub(all)$, which is a violation of update visibility (***U2***). Second, full snapshots are expensive, because they require making copies whenever any initiator writes to the public state. Instead, Maxoid minimizes performance overhead for the normal initiator mode.

Naming of resources in different views. When a delegate B^A updates a resource in public state, Maxoid forks the resource, keeping both the original and the updated versions of the resource.

- All delegates of A see only the updated version with the original name, as part of $Pub(x^A)$. This guarantees delegates that they will read their writes.
- A sees both versions. The original version keeps the original name, as part of $Pub(all)$. The updated version is given a different name, as part of $Vol(A)$.

Commit and clean-up. Data in the volatile state can be retrieved by the initiator with names in a special pattern, i.e., a "tmp" in the path name or the URI. Often, the initiator A (e.g., Dropbox) only wants B^A (an editor) to change one or a few files, but B^A may also generate side effects like cached copies and metadata saved to databases. The desired and undesired changes to public state by B^A all belong to $Vol(A)$. A can selectively **commit** the desired change by copying it from $Vol(A)$ to a non-volatile place. After that, A can discard the entire $Vol(A)$ conveniently because of the fixed naming pattern, to clean up undesired changes. The commit operation can be done by the user manually, or by adding functionality to the initiator for a better user experience.

3.2.4 IPC and initiator policy specification

Android's inter-process communication is based on the native Binder IPC. However, the direct use of it is typically for intra-app, and app-to-system-service communications. Background inter-app communication using content providers is also based on Binder.

In Maxoid, direct Binder IPC for a delegate is restricted to its initiator, other delegates of the same initiator, and trusted system processes.

Intent. Inter-app invocation is done with a higher-level API, **intent**. An app uses an intent to invoke another app: the intent describes an invocation and is passed to Activity Manager Service (via Binder IPC), which finds the suitable target app component and routes the intent to it. The intent itself may contain the sender's sensitive data, or a URI/path name to some sensitive data.

Invocation-transitivity. When B^A invokes app C, the invoked instance is forced to be A's delegate, i.e., C^A. Therefore, B^A cannot leak data in $Priv(A)$ via IPC; it can only invoke A or delegates of A (***S1***). Also, since Maxoid does not stop the invocation, B^A is not disrupted (***U3***). Similarly, broadcast intents from B^A are only delivered to A and delegates of A.

If initiator C invokes app B, the invoked instance can only be either B on behalf of itself or B^C; C cannot invoke B^A to steal $Priv(A)$ from the result of the invocation (***S1***).

Specifying invocation type. When initiator A invokes another app, it can specify whether the invoked app will be started normally (on behalf of itself) or as a delegate of A. If an invocation contains or points to A's data that A thinks needs protection, it should invoke the target app as a delegate. Maxoid has two ways for an initiator to specify this intention, and the details will be discussed in Section 3.5.1.

Maxoid also allows the user to start a delegate B^A without A's explicit invocation if this is the user's intention. The user can specify this intention with the user interface of the system's Launcher (Section 3.5.3).

Maxoid does *not* support **nested delegation**. If B^A specifies to invoke C as B's delegate, that invocation will fail, because B^A can only invoke delegates of A.

3.2.5 Maxoid system architecture

The system architecture of Maxoid is shown in Figure 3.5. It has new components in Android's Activity Manager Service and kernel to track the context of apps (e.g., what initiators they run on behalf of) and intent IPC between them, and choose the correct context for a new invocation (Section 3.2.4, Section 3.5.2). Other components implement Maxoid view switching for file system (Section 3.3) and system content providers (Section 3.4). Zygote is the parent process in Android that forks all app processes, which preloads common Java classes and resources, to speed up application launching.

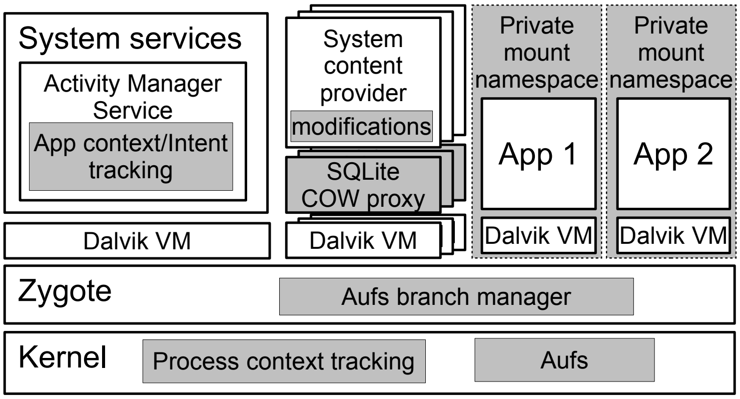


Figure 3.5: Maxoid system architecture. Gray boxes are new components or modifications to Android.

3.3 File system

This section explains how Maxoid manages different views of the file system.

3.3.1 Files in Maxoid views

An app can access private and public files in the same way as it does in Android. It uses regular path names, and Maxoid achieves security transparently by presenting it the correct view of files. In addition, an initiator A's volatile state $Vol(A)$ is a new concept in Maxoid, and files in it can be located by A in a `tmp` directory under the mount point.

Figure 3.6 illustrates a scenario involving A, B^A and another app X, which all read/write some files. Each of them has its own view of these files. Files in $Pub(all)$ are visible to all three app instances, and they have the same view of these files, until B^A's write causes unilateral copy-on-write. B^A can

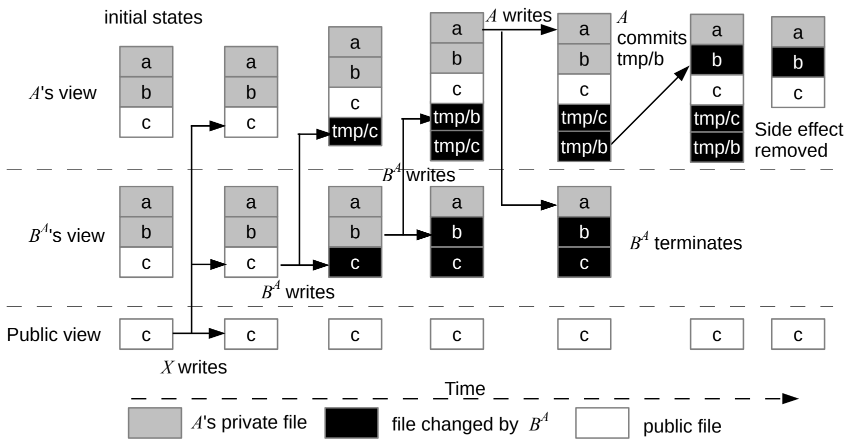


Figure 3.6: Views of files for A, B^A and X. The figure shows a scenario where A wants B^A to edit a file b, but B^A also has side changes on file c.

access files in $Priv(A)$, but any write operation also causes copy-on-write. After B^A writes, Maxoid presents it the updated version with the original path name to let it read its write, while A sees the updated version in the `tmp` directory which is part of $Vol(A)$. X cannot learn any update made by B^A, or any private file of A.

3.3.2 Implementing Maxoid views with Aufs

Aufs[4] is a union file system that can provide a merged view of multiple branches (directories) in a single mount point. If multiple branches contain the same path name, Aufs presents the file in the branch with highest priority. If only that branch is writable, the process' writes are sandboxed in it; modifying a file which does not exist in the writable branch will result in copying that

[4] `http://aufs.sourceforge.net/`

file to the writable branch. Therefore, we can use Aufs to implement per-file copy-on-write.

Maxoid uses the Linux mount namespace to present different views to different apps. When the app process is created, Maxoid first calls `unshare()` in Zygote to create the process' private mount namespace. Maxoid adds an **Aufs branch manager** (Figure 3.5) in Zygote, which selects and mounts the relevant branches for a new app process.

Internal private directory. Maxoid uses a file system-based solution for various types of private state, since shared preferences and private databases are represented as private files. Android assigns each app a private data directory in internal storage, under `/data/data/`. We retain this interface as the private state of an initiator (e.g., $Priv(A)$) or the normal private state of a delegate (e.g., $nPriv(B^A)$).

When B^A starts, the branch manager mounts Aufs at the location of B's private directory as $nPriv(B^A)$, with two branches. One branch is read-only, which is the normal private data directory that the app uses when not running as a delegate; the other branch is writable, which is a directory only accessible to this delegate. The writable private branch has higher priority and is initially empty, thus all writes are redirected to it. The directory of the writable branch is located in a path that only root can directly access; the delegate can only use it via the Aufs mount point.

Aufs is not used for initiators' private directories. B can directly write

$Priv(B)$. However, $Priv(B)$ is a branch of $Priv(B^A)$, and updates to $Priv(B)$ are visible to B^A; if B and B^A run simultaneously, B^A would likely observe inconsistencies in $Priv(B^A)$. To avoid inconsistency without creating full snapshot of $Priv(B)$ or adding overhead to B, a running instance of B will be killed when B^A is invoked.

As discussed in Section 3.2.2, a delegate may also have persistent private state ($pPriv$). It is represented as another directory in internal storage under `/data/data/ppriv`. B^A and B^C use the same path name for persistent private state, but Maxoid presents them different views of this directory by mounting independent Aufs branches at this location. For each delegate, a single writable branch is used.

External storage. Files in external storage, such as an SD card, are world-accessible in Android. External storage is mounted at a public directory, such as `/storage/sdcard`. The mount point varies in different devices, and we use `EXTDIR` to denote it.

Naming volatile files. **Volatile files** caused by delegates' writes to external storage are located in the `tmp` subdirectory under `EXTDIR`. Specifically, if a delegate writes to a file `EXTDIR/`⟨`path`⟩, the corresponding volatile copy can be located by the initiator via path name `EXTDIR/tmp/`⟨`path`⟩. Different initiators have different views of `EXTDIR/tmp`.

Allow private files on external storage for backward compatibility. Currently, Android apps, e.g., Dropbox, often store their files on public external

Mount point	**Branches for A**	**Branches for B^A**
`EXTDIR`	`pub` (rw)	`A/tmp` (rw)
		`pub`
`EXTDIR/data/A`	A/data/A (rw)	`A/tmp/data/A` (rw)
		`A/data/A`
`EXTDIR/data/B`	N/A	`B-A/data/B` (rw)
		`B/data/B`
`EXTDIR/tmp`	`A/tmp` (rw)	N/A

Table 3.2: Aufs mount points for A and B^A. A and B each specify `EXTDIR/A` and `EXTDIR/B` as a private directory on external storage storage. “rw” means a read-write branch, and other branches are read-only.

storage to allow other apps to open them, giving up protection. With Maxoid, Dropbox could store those files in private state and still allow delegates to open them safely. To support such apps without changing their source code, and to avoid using too much space on internal storage (which has limited capacity in many devices), we allow an app A to specify a list of **private directories on external storage** as part of $Priv(A)$.

However, we cannot make a directory private to A by simply disallowing other apps access to it, because apps with access to external storage expect to have access to all files on it. Instead, A and other apps have different views of this directory. Other apps can still use it as a public directory, but only A and its delegates can see A's private files in it.

The Aufs branch manager divides the external storage into different branches (subdirectories): a public branch for all apps, and a private branch for each initiator or delegate. Then it mounts Aufs to `EXTDIR`, using relevant branches. Table 3.2 shows the mount points for A and B^A. Suppose A and B each specify `EXTDIR/data/A` and `EXTDIR/data/B` as a private directory, then

- Files in $Pub(all)$ are located in `pub` branch.
- `EXTDIR/data/A` for A is backed by its private branch `A/data/A`.
- Except `EXTDIR/data/A` and `EXTDIR/tmp`, A accesses files in other places on `pub` branch.
- B^A can read A's private files in `EXTDIR/data/A`, because `A/data/A` is a read-only branch for it.
- B^A's writes to `EXTDIR/data/B` are redirected to branch `B-A/data/B`, which is not visible to A or B.
- B^A's writes to other places are redirected to branch `A/tmp`, which are only visible to A (as $Vol(A)$) and delegates of A (as $Pub(x^A)$). This allows A to get the results of B^A's edits, without letting B^A directly overwrite the original version.

Internal private files exposed to delegates. Maxoid allows a delegate to access its initiator's private data directory in internal storage. We adopt a similar approach as for external storage. To the delegate, the internal directory is part of its view of public state; if it makes modifications, its initiator will see both the original and modified versions, where the modified versions are part of the initiator's volatile state.

Maxoid mounts Aufs for the delegate, with the initiator's private directory as a read-only branch, and a `tmp` directory as a writable branch. We modify Aufs to always allow read access, to allow the delegate to read the read-only branch (the delegate and its initiator have different UIDs); this is safe because Maxoid only mounts Aufs when read is allowed, and an app's process can no longer mount Aufs after Zygote drops root privilege. Similarly, the `tmp` directory is made accessible to the initiator as an Aufs mount.

3.4 System content providers

We describe the views of data in system content providers that Maxoid presents to apps.

3.4.1 System content providers in Maxoid

System content providers, like Downloads, Media, Contacts and Calender, are built-in packages that provide standard platform-level APIs. They typically use SQLite databases as backends. We built a copy-on-write proxy layer (Section 3.4.2) on top of SQLite, and modify these providers to use the proxy so that they can switch views for different app instances.

User Dictionary is a simple system content provider that maps URIs to records in the user dictionary database, the columns of which include ID, Word, Frequency, etc. A record with ID=`n` can be retrieved via URI `content://-user_dictionary/words/n`. URI `content://user_dictionary/words` represents all records in the database.

The ID column is the primary key in the database. This type of URI-to-ID mapping is generic for many system content providers, including Downloads and Media. Essentially, a URI is mapped to a database row (or a group of rows). Our proxy layer implements per-row, per-initiator unilateral copy-on-write, and thus can naturally support these system content providers with minimal code change.

In Maxoid, the results of write operations (insert or update) by a delegate B^A are stored as **volatile records**, as part of $Vol(A)$. B^A cannot overwrite any public records. Similarly, when B^A deletes a URI, the public record is not affected; instead, Maxoid emulates a deletion for B^A by creating a "whiteout" volatile record (Section 3.4.2). For each ID, there is at most one volatile record in $Vol(A)$. If the volatile record for ID=`n` doesn't exist, B^A sees the public record (if it exists) in the result of a query. After the volatile record is created by a delegate's insert or update, any operation from B^A on ID=`n` will happen on the volatile record.

B^A's view of the content provider is transparent. B^A always uses normal URIs. It only sees a single version for each ID and can read its own writes. On the other hand, if A uses a normal URI, the content provider will operate on the public records; to access volatile copies, it can use **volatile URIs**, which has a `tmp` component, e.g.,

- `content://user_dictionary/tmp/words/⟨n⟩`
- `content://user_dictionary/tmp/words/`

for a specific ID and all volatile records respectively.

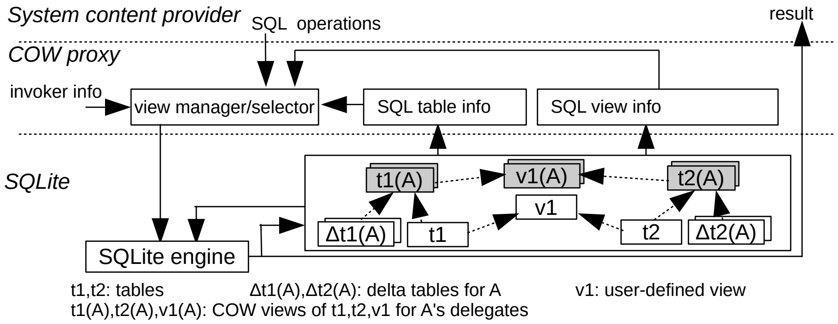


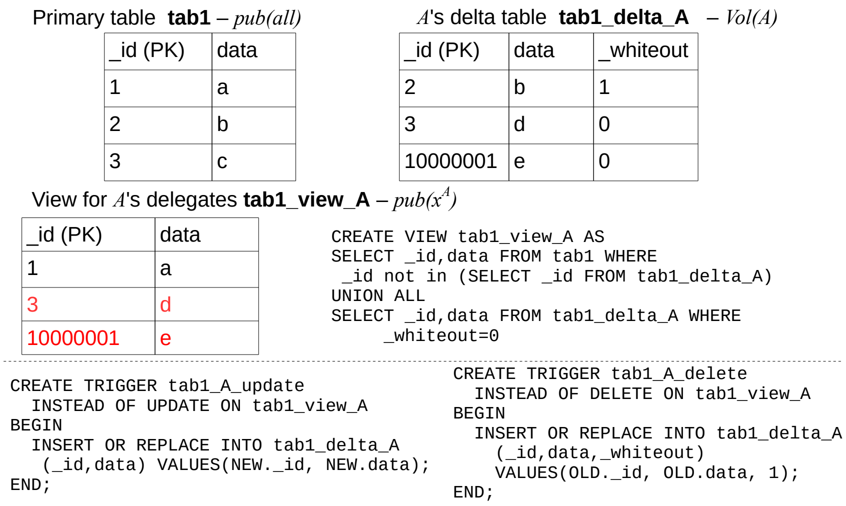
Figure 3.7: COW proxy interacts with the content provider and SQLite. Note that `v1` is an SQL view defined by the content provider.

3.4.2 SQLite copy-on-write proxy layer

We built a copy-on-write (COW) proxy layer on top of SQLite API, to minimize modifications to content providers.

Figure 3.7 shows how the proxy layer interacts with the content provider and SQLite. It provides the same APIs as SQLite to content providers for normal database operations, and some additional APIs for administrative operations. It achieves unilateral per-name copy-on-write (Section 3.2.3), where a name corresponds to a database row.

We call each table defined by the content provider a **primary table**. Primary tables only store data that belongs to $Pub(all)$. For each primary table, the proxy maintains per-initiator **delta tables**, to store volatile state of different initiators. We say a **COW view** for A's delegates is the view of a specific primary table in $Pub(x^A)$. A COW view is implemented as a **SQL view** – a virtual table based on a query result in SQL – defined on the primary table and the delta table.



Primary table **tab1** – *pub(all)*

_id (PK)	data
1	a
2	b
3	c

A's delta table **tab1_delta_A** – *Vol(A)*

_id (PK)	data	_whiteout
2	b	1
3	d	0
10000001	e	0

View for *A*'s delegates **tab1_view_A** – *pub(x^A)*

_id (PK)	data
1	a
3	d
10000001	e

```
CREATE VIEW tab1_view_A AS
SELECT _id,data FROM tab1 WHERE
 _id not in (SELECT _id FROM tab1_delta_A)
UNION ALL
SELECT _id,data FROM tab1_delta_A WHERE
      _whiteout=0
```

```
CREATE TRIGGER tab1_A_update
  INSTEAD OF UPDATE ON tab1_view_A
BEGIN
  INSERT OR REPLACE INTO tab1_delta_A
   (_id,data) VALUES(NEW._id, NEW.data);
END;
```

```
CREATE TRIGGER tab1_A_delete
  INSTEAD OF DELETE ON tab1_view_A
BEGIN
  INSERT OR REPLACE INTO tab1_delta_A
    (_id,data,_whiteout)
    VALUES(OLD._id, OLD.data, 1);
END;
```

Figure 3.8: Delta table and the view for delegates maintained by the SQLite proxy layer.

Per-initiator delta tables and COW views. A delta table has all columns in the primary table, plus an additional boolean column called `_whiteout` (Figure 3.8). When the content provider queries for B^A, the result will be generated from both the primary table and A's delta table. If a row R_d in the delta table has the same primary key as a row R_p in the primary table, R_p will not appear in the result. If R_d has `_whiteout`=0 and satisfies the `WHERE` conditions in the query, it will be included in the result. `_whiteout` is thus an indicator of whether the record has been deleted for delegates; if R_d has `_whiteout`=1, the result will include neither R_d nor R_p.

The proxy implements the table's *COW view* for an initiator's delegates, based on a *SQL view*. The COW view is transparent, which means it can be used in the same way as a regular table, and can be contained in the definition of other SQL views. It is defined as the compound `SELECT` statement using `UNION ALL` in Figure 3.8. Its definition satisfies the constraints for SQLite's subquery flattening optimization[5], which makes queries on it efficient because the query planner moves the `WHERE` clause (if any) on this view into the two inner subqueries.

However, SQLite views are read-only. To support insert, the proxy places B^A's inserts into the delta table. Typically, the primary key is generated by incrementing the current maximum primary key in the table. The primary table's primary key starts from 1. To avoid naming collision, the delta table's primary key starts at a large number N for newly inserted rows.

To support update and delete, we define `INSTEAD OF` triggers on the per-initiator COW views (Figure 3.8). These triggers implement per-row copy-on-write, which confines modifications in the delta table.

Delta tables and COW views are created on demand. A's delta table and COW view are created when the first volatile record is created, by either A itself or its delegates.

User-defined SQL views. The user of SQLite, i.e., content providers in thie case, may define their own *SQL views* over base tables. The proxy maintains delta tables only for base tables, not for SQLite views which are stateless. But to support user-defined SQL views, the proxy maintains per-initiator COW

[5] `http://www.sqlite.org/optoverview.html`

views for each of them, which are created on demand, and defined identically to the original user-defined SQL views, except that the base tables in the definition are replaced with their corresponding COW views. Moreover, one user-defined SQL view may use another user-defined SQL view as one of its "base tables"; accordingly, the proxy maintains a hierarchy of COW views (Figure 3.7), and the user-defined view's COW view can only be created after the COW views of its base tables are created.

Maxoid view selection. The COW proxy uses a Maxoid API to get the information about the calling process, which tells whether the caller is a delegate and what its initiator is. It then selects the correct Maxoid view. If the caller is not a delegate, the operation will only involve primary tables as normal; otherwise, the proxy selects the correct delta tables or COW views, and creates them if they do not exist.

Additionally, the proxy allows the content provider to select what Maxoid view it would like to use. This enables the content provider to do administrative operations and implement new URIs for volatile state. The proxy defines an administrative view, which contains data in the primary table and all delta tables, with an additional column that indicates what state a row belongs to.

3.4.3 Modifications to content providers

So far, we have ported three system content providers using the COW proxy: User Dictionary, Downloads, and Media.

User Dictionary. User Dictionary is purely a passive storage service, which means it only queries/updates data when a client explicitly requests so. In this case, porting is trivial, though we add new URIs for volatile state.

Downloads. Although a delegate is not supposed to access the network, we modify Downloads to allow an initiator to create volatile downloads, e.g., for incognito mode. Downloads has not only storage, but also background threads for downloading files and mechanisms to generate notifications. They actively query and update data. Thus it needs to use the administrative view to get all public and volatile records, and track what state a record belongs to. Downloads has two tables, `downloads` and `request_headers`. For a delegate's operation, the proxy selects the corresponding views for both tables. For operations by Downloads itself, Downloads selects the correct view based on the information it tracks. Downloads stores the path names of downloaded files in its database, and needs to access those files. Maxoid makes all volatile `tmp` directories visible to Downloads, but the path names of the files are different from those stored in the database (which are transparent to clients). We wrote a wrapper of Java's `File` class to automate locating files.

Media. Media defines multiple SQL tables and views. For example, it stores data for different types of media files in a single base table called `files`; `images`, `audio_meta` and `video` are views defined as selections over `files`. `audio` is a view defined on three tables/views, including `audio_meta`. We use the COW proxy to manage the hierarchy of COW views. Like Downloads, Media also has extra services beyond data storage, e.g., creating thumbnails. Similarly, modified Media keeps track of what state a record/request belongs to.

3.5 API and implementation

Our Maxoid prototype is based on Android 4.3.2.

3.5.1 API summary

Maxoid introduces a few new (sometimes optional) changes for initiators. For delegates, although Maxoid is mostly transparent, it defines new optional APIs for better usability.

APIs for initiators.

1. An app can specify a list of private directories in external storage (Section 3.3.2) via an XML file called the **Maxoid manifest**.

2. When the initiator invokes another app, it can specify whether the invoked app will be a delegate of it in two ways:

1) A new flag in `Intent`*.* When this flag is set, the invoked app will be a delegate. App developers can modify their code to use this flag when

Maxoid is available.

2) Intent filters for invokers. Maxoid allows an app to specify a whitelist or blacklist of intent filters in its Maxoid manifest. When the initiator sends an intent, Maxoid checks it against the filters to decide whether the invoked app should be a delegate. Code change is not needed for initiators.

Additionally, we also modify the system's launcher, to allow B^A to start without A's explicit invocation if this is the user's intention (Section 3.5.3).

3. An initiator can manage its volatile state (Section 3.3 and Section 3.4).

4. When an initiator creates a new record in a system content provider, Maxoid allows it to specify whether this record is volatile or not. By default, the new record will be public; if it asserts the `isVolatile` flag in the `ContentValues` parameter for this insert call, the new record will be created in its volatile state. This API can help a browser to implement incognito download (Section 3.6).

APIs for delegates. First, Maxoid introduces persistent private state, which is a directory in internal storage (`/data/data/ppriv/`⟨`package_name`⟩) (Section 3.2.2, Section 3.3.2). Second, an app can query whether it runs as a delegate, and what initiator app it runs on behalf of.

Note that Maxoid does not support nested delegation. An app can only make private invocations or create its own volatile records when it is an initiator.

3.5.2 Tracking app execution context

Section 3.3 and Section 3.4 already cover implementation of Maxoid views for file system and system content providers. This section discusses how Maxoid tracks whether an app is running normally or on behalf of others, which requires modification to the following system components.

1. Activity Manager Service. A delegate can only make normal invocations which make the invoked apps also delegates of the same initiator (invocation-transitivity in Section 3.2.4). If an initiator invokes another app, Maxoid checks the flag in the intent and the intent filters to decide whether it invokes a delegate. (Currently, if the invoked app already has an instance running, but not on behalf of the current initiator, that instance will be killed.) An intent's direct destination may be a system component, like `ResolverActivity` which shows a list of candidate apps when the user opens a file. In this case, `ResolverActivity` is considered as an intent channel rather than an app instance. When Activity Manager Service starts a new activity, Maxoid passes information about the app and its initiator to Zygote.

2. Zygote. When forking a new process, Zygote checks the parameters and passes them to the kernel sysfs interface. It manages Aufs branches and mounts Aufs in the process' mount namespace to switch views of the file system (see Section 3.3).

3. Kernel. 1) We add a sysfs interface for Zygote to communicate app

and initiator information to the process' `task_struct`. 2) Maxoid emulates loss of network connection for delegates by returning error code `ENETUNREACH` in the `connect` system call (similar to AppFence [HHJ$^+$11]). 3) Direct Binder IPC for a delegate is restricted to trusted system services and system content providers, its initiator and delegates of the same intiator.

4. System content providers. We modified 3 system content providers (User Dictionary, Downloads and Media) to support Maxoid (see Section 3.4). In addition, to fully disable a delegate's network access, returning an error code in `connect` is not sufficient, because a delegate may request Download Provider to fetch files from the web for it, potentially leaking sensitive data via the requested URL. Therefore, Maxoid also emulates a network error in Download Provider for download requests from delegates. Nonetheless, a delegate may still add or update entries in the database for existing files, because that does not access network.

5. Other system services. Bluetooth Manager Service and Telephony Provider are modified to prevent delegates from sending data via Bluetooth or SMS services. Clipboard Service is modified to create separate clipboard instances for delegates.

3.5.3 User interface

We modify the system's Launcher to improve usability. 1) The user may start a delegate on behalf of an initiator, without the initiator invoking it. For instance, Maxoid allows the user to start Camera as Email's delegate by dragging Email's icon into an "Initiator" drop target before clicking Camera's icon. 2) By dragging the icon of A into a "ClearVol" drop target, the user can clear the volatile state of A. 3) By dragging the icon of A into a "ClearPriv" drop target, the user can clear $Priv(x^A)$ for all x.

3.6 Maxoid use cases

Out of the 77 data processing apps we analyzed in Section 3.1, only three (DocuSign, EasySign and ThinkTI Document Converter) cannot work when they run as delegates, due to loss of network connection. We describe five use cases of Maxoid, where the first four secure initiators to use those unmodified data processing apps, and the last improves the delegate's usability with minimum code change.

Securing Dropbox. Dropbox stores files on a directory in external storage. We use the Maxoid manifest to specify this directory to be private, and a filter saying that any intent from Dropbox with `VIEW` action (indicating the user clicking a file) is private, i.e., to invoke a delegate. Thus, other apps cannot see the files unless invoked by the user clicking a file from Dropbox. Dropbox sees the delegates' modifications under `EXTDIR/tmp`. Without modifying Dropbox's source code, we require the user to manually upload the modified file if it is desired, from `EXTDIR/tmp`. After that, the user can clear Vol(Dropbox) to remove any undesired changes.

Even though Dropbox does not invoke camera apps, the user can start a camera app as Dropbox's delegate using the Launcher (Section 3.5.3), and take a private photo for Dropbox.

Securing Email attachments. We use a filter to specify that `VIEW` intents are private. As a result, when the user clicks the "VIEW" button on the attachment, the invoked app will be Email's delegate. (The user can still intentionally save the file to external storage and Downloads Provider, by clicking the "SAVE" button.)

The user can also start an app via Launcher as Email's delegate without Email invoking it.

Enhancing Browser's incognito mode. The Browser app uses Android's DownloadManager API (a wrapper of Downloads Provider's API) to download files. We extend this API to allow an initiator to specify whether a requested download from it should be stored in the public state or its volatile state. Then, we add 1 line of code for Browser, such that downloads from an incognito tab are stored in the volatile state, while downloads from a normal tab are stored in public state. When the user clicks a download complete notification, a proper app will be started as a delegate of Browser if this download is from an incognito tab. This functionality is supported by our Downloads Provider. The downloaded file, the corresponding entry in Downloads Provider, and any updates by the delegate depending on this download will be discarded when the user clears $Vol(\text{Browser})$ and $Priv(x^{\text{Browser}})$. To extend incognito mode to a QR code reader app, the user can start it as Browser's delegate using the system's Launcher.

Wrapper app. We write an app which does nothing but holding sensitive documents. It can be used as an initiator to force "real apps" into a *system-wide incognito mode* by clearing the volatile state after use.

Using delegates' persistent private state. Maxoid supports unmodified delegate apps. As discussed in Section 3.2.2, delegate apps that are aware of Maxoid can also be modified for better usability. EBookDroid[6] is an open-source app for viewing and managing documents. It stores recent documents and bookmarks in its private database. We modify 45 lines of code to make use of the persistent private state. When it runs normally, it stores new entries for recent files or bookmarks to a database in $nPriv$; when it runs as a delegate, it stores new entries in $pPriv$, and shows a list of recent files merged from both $nPriv$ and $pPriv$.

3.7 Performance

We measure performance overhead added by Maxoid, on a Nexus 7 tablet, which has 2GB of DDR3L RAM and 1.5GHz quad-core Qualcomm Snapdragon S4 Pro CPU, and runs Android 4.3.2. Maxoid barely adds any

[6] `https://code.google.com/p/ebookdroid/`

Setup	**CPU-bound operations**	**Internal File System**					
		4KB files			1MB files		
		read	write	append	read	write	append
initiator	0	0					
delegate	0	7.5%	31.7%	58.7%	4.8%	18.1%	52.8%

Table 3.3: Microbenchmark overheads for CPU-bound operations and file system, compared to Android. Read – read files; Write – create and write to files; Append – append to the original files to double their sizes.

Setup	**User Dictionary Provider**				
	insert	update	query 1 word	query 1k words	delete
initiator	1.3%	0.4%	0.5%	0.2%	1.0%
delegate	8.1%	16.1%	5.6%	13.7%	17.3%

Table 3.4: Microbenchmark overheads for User Dictionary Provider, compared to Android. Size of table: 1000 rows. Query 1 word is done by specifying the word ID in the URI; query 1k words is selecting all words in the database.

overhead to initiators. For delegates, Maxoid does not add overhead for CPU-intensive computations, only for I/O operations, i.e., file and content provider operations.

3.7.1 Microbenchmarks

CPU-bound operations. We measure the time for performing matrix multiplications. Maxoid adds no overhead to initiators and delegates, compared to unmodified Android.

File system. Maxoid uses a single branch at any internal or external mount point for initiators, thus incurs no overhead for initiators. However, it uses two branches at each internal or external mount point for delegates, except the persistent private state. We measure the performance of Aufs for delegates, on a microbenchmark app that uses its internal file storage. The results are shown in Table 3.3. We test operations including read, write and append. Before append operations for delegates, the original files are are on a read-only branch, and the append operations copy them to the writable branch, resulting in large overhead. However, the overhead could be reduced if a block-level copy-on-write file system (as opposed to file-level) were used; we choose Aufs for features that ease our prototype development.

User Dictionary Provider. We measure the slowdown for content provider operations, using the User Dictionary Provider as an example. The slowdowns for both initiators and delegates are shown in Table 3.4. The baseline is an unmodified Android OS. Slowdowns for the initiator are negligible. For delegates, updates are executed before there are entries in the delta table, so that copy-on-write will happen; queries are executed after updates, so that both primary and delta tables will be involved. Maxoid adds less than 18% overhead for delegates.

3.7.2 Macrobenchmarks

Download and Media Providers. We measure the time for 1) downloading 100 1KB files, using `DownloadManager`, and 2) scanning 100 image files and storing the metadata to Media Provider. Table 3.5 shows the result, where

Setup		Android	Maxoid	
			to public state	to volatile state
Time (s)	download	7.29±0.39	7.13±0.28	7.23±0.21
	image	1.54±0.02	1.54±0.02	1.55±0.02

Table 3.5: Times for 1) downloading 100 1KB files, and 2) scanning 100 780KB image files and storing the metadata to Media Provider.

the baseline is an unmodified Android. For Download Provider, our tester app can request the downloaded files to be saved in either public or volatile state; in both cases, the tester app runs as an initiator to access the network. For Media Provider, the tester app first runs as an initiator to store metadata into public state, then runs as a delegate to store metadata into volatile state. The overhead is negligible for all cases.

Application benchmarks. We measure the latency of performing several application-specific tasks, as listed in Table 3.6. Our experiments show that Maxoid's impact on user-perceivable latency of these tasks is very small. This is because the typical usage of many mobile apps does not involve data-intensive operations, and Maxoid does not add overhead to UI-related and CPU-intensive workload. For example, the time for reading a 1.6 MB PDF file is negligible compared to the time for rendering it.

App	**Task**	**Latency (ms)**		
		Android	Maxoid	
			Initiator	Delegate
Adobe Reader	open a 1.6 MB file	1213±27	1207±20	1221±14
	in-file search	3206±57	3218±80	3197±50
CamScanner	process a scanned page	7338±323	7420±298	7446±249
CameraMX	take a photo	1214±41	1251±44	1255±90
	save an edited photo	1829±89	1855±59	1897±73

Table 3.6: User-perceivable latency of performing various tasks using different apps.

3.8 Discussion

We discuss the applicability of Maxoid's model to other mobile platforms, and the limitations of Maxoid.

3.8.1 Applicability to other platforms

The state model of Maxoid applies to app-centric platforms, which treat apps as different principals. Such platforms provide storage abstractions for both private and public storage, where private data can only be accessed by the owning app, and public data are shared by apps. For example, like Android, **Windows Phone 8** assigns each app an isolated private directory, and exposes external storage as a shared resource subject to coarse-grained access control. Similarly, **iOS** provides each app a private directory for file storage; it does not have a shared file system, but instead provides high-level, device-wide shared resources such as photos and contacts. **FireFox OS** is a platform that runs mobile apps written in Web code; apps have private storage options such as IndexedDB, and share public resources like the SD card and contacts.

In principal, Maxoid's model is generic and can be used in all those platforms. However, implementing the model would be platform-specific. Maxoid leverages Android's unified data abstractions – files and content providers – to minimize modifications. Since iOS does not provide a shared file system, the Maxoid-style multi-branch external storage solution is unnecessary; on the other hand, different techniques would be needed to support volatile entries in the photo gallery.

3.8.2 Scope and limitations

Use cases. Maxoid is targeted at cases where delegates are short-lived foreground tasks, so network disruption and state divergence are not likely to cause usability issues. Maxoid does not support scenarios where delegates need to send initiators' private data to remote servers for processing. Maxoid is an incremental improvement over Android; it provides better security for its target use cases, while maintaining Android's legacy behavior for unsupported use cases, instead of breaking them.

Code changes. Maxoid needs code changes to system content providers, though with the help of the SQLite proxy. Content providers often involve specific tasks that are not generic enough to be supported in a unified way. Maxoid is not totally transparent to initiators, because the concept of volatile state is new. However, the API is simple enough to allow small or no modifications to initiators in many cases, enabling security enhancements that cannot be achieved in Android.

App-defined content providers. As opposed to system content providers, app-defined content providers are not considered shared resources. They are often backed by private files or databases, which Maxoid treats as the private state of their owning apps. Communicating among apps with content providers can be considered declassification, so Maxoid does not support per-URI volatile copies for app-defined content providers. In Android, IPC with content providers is implemented using the low-level Binder interface, and Maxoid's restrictions on Binder IPC prevents delegates from leaking data (Section 3.2.4). Modifications to data in delegate-defined content providers would be discarded by Maxoid eventually. However, initiators are responsible for auditing write requests to their content providers if they want to avoid unauthorized modifications. For example, the built-in Email app has a content provider for attachments, but it only grants temporary, read-only access for an entry to a document viewer on an explicit invocation; the document viewer would need to create a copy of the attachment if the user saves changes, which is the behavior of Adobe Reader.

Chapter 4

Earp: abstraction and protection for structured data

Modern mobile apps communicate and exchange data with other apps almost as much as they do with the operating system. Many popular apps now occupy essential places in the app "ecosystem" and provide other apps with services, such as storage, that have traditionally been the responsibility of the OS. For example, an app may rely on Facebook to authenticate users, Google Drive to store users' data, WhatsApp to send messages to other users, Twitter to publicly announce users' activities, etc. In platforms like Android, even some standard APIs are actually implemented in built-in apps (Section 2.3), like the photo gallery, calendar and contacts.

Traditionally, operating systems have provided abstractions and protection for storing and sharing data. The data model in UNIX is byte streams, stored in files protected by owner ID and permission bits and accessed via file descriptors (Section 2.1). UNIX has a uniform access-control model for both

This chapter is based on previous publication [XHK$^+$16] "Earp: Principled Storage, Sharing, and Protection for Mobile Apps", by Yuanzhong Xu, Tyler Hunt, Youngjin Kwon, Martin Georgiev, Vitaly Shmatikov and Emmett Witchel, in the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI), Santa Clara, CA, March 2016. My contributions to this publication include designing and implementing the protection mechanisms for relational data, the descriptor-based APIs, the object graph library, and the inter-app service framework.

storage and inter-process communication: users specify permissions on files, pipes, and sockets, and the OS dynamically enforces these permissions.

Modern mobile platforms provide higher-level abstractions to manage *structured data*, and relational databases have become the de facto hubs for apps' internal data [SBL$^+$14]. These abstractions, however, are realized as app-level libraries. Platform-level access control in Android and iOS inherits UNIX's coarse-grained model and has no visibility into the structure of apps' data. Today, access control in mobile platforms is a mixture of basic UNIX-style mechanisms and ad hoc user-level checks spread throughout different system utilities and inter-app services. Apps present differing APIs with ad hoc access-control semantics, different from those presented by the OS or other apps. This leaves apps without a clear and consistent model for managing and protecting access to users' data and leads to serious security and privacy vulnerabilities.

In Chapter 3, we have treated some of Android's built-in content provider apps (e.g., Media and Downloads) as part of the platform, but also have to modify each of them individually to support Maxoid's security model. This is precisely because Android does not have a platform-level abstraction for high-level structured data, such that we could not achieve fine-grained protection at the platform level. We now consider them regular apps, and investigate the feasibility of using a common abstraction for them.

In this chapter, we explore the benefits and challenges of using the relational model as the unified, platform-level abstraction of structured data. We design, implement, and evaluate a prototype of Earp, a new mobile platform that uses this model for both storage and inter-app services, and demonstrate that it provides a principled, expressive, and efficient foundation for the data storage, data sharing, and data protection needs of modern mobile apps.

First, we demonstrate how apps can use the relational model not just to define data objects and relationships, but also to *specify access rights directly as part of the data model.* For example, an album may contain multiple photos, each of which has textual tags; the right to access an album confers the right to access every photo in it and, indirectly, all tags of these photos.

Second, we propose a *uniform, secure data-access abstraction* and a new kind of reference monitor that has visibility into the structure of apps' data and can thus enforce fine-grained, app-defined access-control policies. This enables apps to adhere to the principle of least privilege [Sal74] and expose some, but not all, of users' private data to other apps. App developers are thus relieved of the responsibility for writing error-prone access-control code. The unifying data-access abstraction in Earp is a *subset descriptor.* Subset descriptors are capability-like handles that enable the holder to operate on some rows and columns of a database, subject to restrictions defined by the data owner. Our design preserves efficiency of both querying and access control.

Third, we implement and evaluate a prototype of Earp based on Firefox OS, a browser-based mobile platform where all apps are written in Web languages such as HTML5 and JavaScript. The browser-based design enables Earp to conveniently add its data abstractions and access-control protections to the platform layer while maintaining support for legacy APIs. While native platforms like Android have different software architectures that are not ideal to build a fully-featured Earp prototype on, we show that some of Earp's core concepts can be applied to them by designing and implementing Earp-style protections to Android's content provider framework.

Fourth, to demonstrate how apps benefit from Earp's structured access control, we adapt or convert several *essential utilities and apps*. We show how local apps, such as the photo manager, contacts manager, and email client, can use Earp to impose fine-grained restrictions on other apps' access to their data. We also show how remote services, such as Google Drive and an Elgg-based social-networking service, can implement local proxy apps that use Earp to securely share data with other apps without relying on protocols like OAuth.

We hope that by providing efficient, easy-to-use storage, sharing, and protection mechanisms for structured data, Earp raises the standards that app developers expect from their mobile platforms and delivers frontier justice to the insecure, ad hoc data management practices that plague existing mobile apps.

4.1 Inadequacy of existing platforms

In today's mobile ecosystem, many apps act as data "hubs." They store users' data such as photos and contacts, make this data available to other apps, and protect it from unauthorized access. The data in question is often quite complex, involving multiple, inter-related objects.

Inadequate protection for storage. The coarse-grained protection for storage (Section 2.1) in existing platforms do not provide adequate support for mobile apps' data management. App developers roll their own and predictably end up compromising users' privacy. For example, Dropbox on Android stores all files in public external storage, giving up all protection. WhatsApp on iOS automatically saves received photos to the system's gallery. When the email app on Firefox OS invokes a document viewer to open an attachment, the attachment is copied to the SD card shared by all apps.

A systematic study [ZJ13] in 2013 discovered 2,150 Android apps that unintentionally make users' data—SMS messages, private contacts, browsing history and bookmarks, call logs, and private information in instant messaging and social apps (e.g., the most popular Chinese social network, Sina Weibo)—available to any other app.

Inadequate protection for inter-app services. Services and protocols that involve multiple apps have suffered from serious security vulnerabilities and logic bugs [WXWC13, XBL+15, SB12, LZX+14, VGN14]. While vulnerabilities in individual apps can be patched, the root cause of this sorry state of affairs is the inadequacy of the protection mechanisms on the existing mobile platforms, which cannot support the principle of least privilege [Sal74].

Existing platforms provide limited facilities for sharing data via inter-app services. Android apps can use *content providers* to define background data-sharing services with a database-like API, where data are located via

URIs. However, permission-based access control for content providers is coarse-grained (Section 2.3). If a service app needs fine-grained protection, writing the appropriate code is entirely the app developer's responsibility. Unsurprisingly, access control for Android apps is often broken [SH14, ZJ13].

Android's URI permission mechanism may be fine-grained, but it is a capability-passing mechanism rather than a policy enforcement mechanism (Section 2.3). The access-control logic still resides in the application itself, making URI permissions difficult to use for programmatic access control. Android mostly uses them to involve the user in access-control decisions, e.g., when the user clicks on a document and chooses an app to receive it.

As also discussed in Section 2.3, iOS apps cannot directly share data via the file system or background services.

Without principled client-side mechanisms for protected sharing, mobile developers rely on server-side authentication protocols such as OAuth that give third-party apps restricted access to remote resources. For example, Google issues OAuth tokens with restricted access rights, and any app that needs storage on Google Drive attaches these tokens to its requests to Google's servers[1]. Management of OAuth tokens is notoriously difficult and many apps badly mishandle them [VGN14], leaving these apps vulnerable to impersonation and session hijacking due to token theft, as well as identity misbinding and session swapping attacks such as cross-site login request forgery [SB12]. In

[1] `https://developers.google.com/drive/android/auth`, `https://developers.google.com/drive/ios/auth`

2015, a bug in Facebook's OAuth protocol allowed third-party apps to access users' private photos stored on Facebook's servers[2].

Inadequate protection model. Protection mechanisms on the existing platforms are based on permissions attached to individual data objects. These objects are typically coarse-grained, e.g., files. Even fine-grained permissions (e.g., per-row access control lists in a database) do not support the protection requirements of modern mobile apps. The fundamental problem is that data objects used by these apps are *inter-related*, thus any inconsistency in permissions breaks the semantics of the data model.

Per-object permissions fail to support even simple, common data sharing patterns in mobile apps. Consider a photo collection where an individual photo can be accessed directly via the camera roll interface, or via any album that includes this photo. As soon as the user wants to share an album with another app, the per-object permissions must be changed for every single photo in the album. Since other types of data may be related to photos (e.g., text tags), the object-based permission system must compute the transitive closure of reachable objects in order to update their permissions. This is a challenge for performance and correctness.

In practice, writing permission management code is complex and error-prone. App developers thus tend to choose coarse-grained protection, which does not allow them to express, let alone enforce their desired policies.

[2] `http://www.7xter.com/2015/03/how-i-exposed-your-private-photos.html`

4.2 Design goals and overview

Throughout the design of Earp, we rely on the platform (i.e., the mobile OS) to protect the data from unauthorized access and to confine non-cooperative apps. Earp provides several platform-enforced mechanisms and abstractions to make data storage, sharing, and protection in mobile apps simpler and more robust.

- Apps in Earp store and manage data using a uniform, relational model that can easily express relationships between objects as well as access rights. This allows app developers to employ standard database abstractions and relieves them of the need to implement their own data management.

- Apps in Earp give other apps access to the data via structured, fine-grained, system-provided abstractions. This relieves app developers of the need to implement ad hoc data-access APIs.

- Apps in Earp rely on the platform to enforce their access-control policies. This separation of policy and mechanism relieves app developers of the need to implement error-prone access-control code.

Efficient system-level enforcement requires the platform to have visibility into the data structures used by apps to store and share data. In the rest of the chapter, we describe how this is achieved in Earp.

4.2.1 Data model

UNIX has a principled approach for protecting both storage and IPC channels, based on a unifying API—file descriptors. On modern mobile platforms, however, data management has moved away from files to structured storage such as databases and key-value stores.

In Earp, the unifying abstraction for both storage and inter-app services is *relational data*. This approach (1) helps express relationships between objects, (2) integrates access control with the data model, and (3) provides a uniform API for data access, whether by the app that owns the data or by other apps.

Unifying storage and services is feasible because Earp apps access inter-app services by reading and writing structured, inter-related data objects via relational APIs that are similar to those of storage. A service is defined by four *service callbacks* (Section 4.4), which Earp uses as the primitives to realize the relational API.

Earp uses the same protection mechanism for remote resources. For example, a remote service such as Google Drive can have a *local proxy* app installed on the user's device, which defines an inter-app service that acts as the gateway for other apps to access Google's remote resources. Earp enforces access control on the proxy service in the same way as it does with all inter-app services, avoiding the need for protocols such as OAuth.

Earp not only makes it easier to manage structured data that is pervasive in mobile apps, but also maintains efficient, protected access to files and directories.

4.2.2 Access rights

All databases and services in Earp have an owner app. The owner has the authority to define policies that govern other apps' access, making Earp a discretionary access control system. The names of databases and services are unique and prefixed by the name of the owner app.

Earp's protection is fine-grained and captures the relationships among objects. In the photo gallery example, each photo is associated with some textual tags, and photos can be included in zero, one, or several albums. Fine granularity is achieved by simple *per-row ACLs*, allowing individual photos to each have different permissions. However, per-object permissions alone can create performance and correctness problems when apps share collections of objects (Section 4.1).

To enable efficient and expressive fine-grained permissions for inter-related objects, Earp introduces **capability relationships**—relationships that confer access rights among related data. For example, if an app that has access rights to an album traverses the album's capability relationship to a photo, the app needs to automatically obtain access rights to this photo, too. Capability relationships only confer access rights when traversed in one direction. For example, having access to a photo does not grant access to all albums that include this photo.

Capability relationships make it easy for apps to share ad hoc collections. For example, the photo gallery can create an album for an ephemeral messaging app like Snapchat, enabling the user to follow the principle of least privilege and install Snapchat with permissions to access only this album (and, transitively, all photos in this album and their tags).

Capability relationships also enable Earp to use very simple ACLs without sacrificing the expressiveness of access control. There are no first-class concepts like groups or roles, but they can be easily realized as certain capability relationships.

4.2.3 Data-access APIs

In Earp, access to data is performed via **subset descriptors**. A subset descriptor is a capability "handle" used by apps to operate on a database or service. The capability defines the policy that mediates access to the underlying structured data, allowing only restricted operations on a subset of this data.

The holder of a subset descriptor may transfer it to other apps, possibly *downgrading* it beforehand (removing some of the access rights). Intuitively, a subset descriptor is a "lens" through which the holder accesses a particular database or service.

Critically, the OS reference monitor ensures that all accesses comply with the policy associated with a given descriptor. Therefore, app developers are only responsible for defining the access-control policy for their apps' data

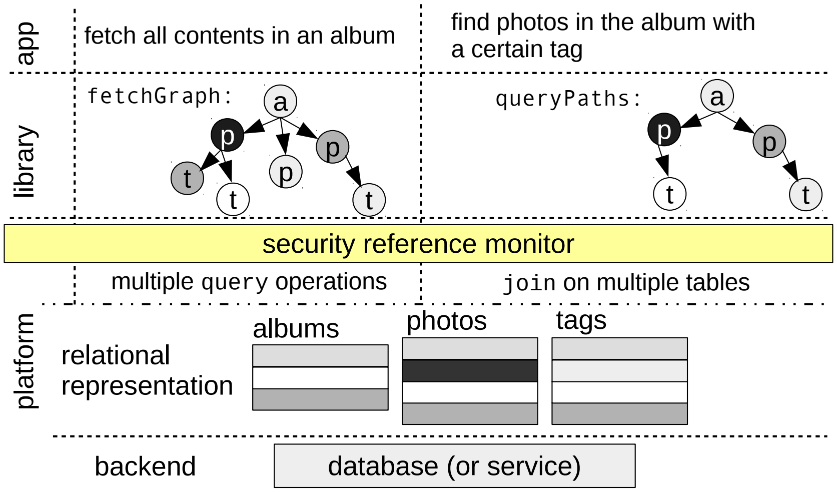

app
fetch all contents in an album
find photos in the album with a certain tag
library
fetchGraph:
queryPaths:
a
p
t
security reference monitor
multiple query operations
join on multiple tables
platform
relational representation
albums
photos
tags
backend
database (or service)

Figure 4.1: Platform- and library-level representations of structured data in Earp.

but not for implementing the enforcement code.

Capability relationships make access rights for one object dependent on other objects. This is a challenge for efficiency because transitively computing access-control decisions would be expensive. To address this problem, apps can *create subset descriptors on demand to buffer access-control decisions for future tasks*. For example, an app can use a descriptor to perform joins (as opposed to traversal) to find all photos with a certain tag, then create another descriptor to edit a specific photo based on the result of a previous join. The photo access rights are computed once and bound to the descriptor upon its creation. Earp thus enjoys the benefits of both the relational representation (efficient joins) and the graph representation (navigating a collection to enumerate its members).

To facilitate programming with structured data, Earp provides a library that presents an *object graph* API backed by databases or inter-app services (see an example in Figure 4.1). This API is functionally similar to the Core Data API in iOS, but each internal node is mapped to a platform-level data object under Earp's protection. This API relieves developers of the need to explicitly handle descriptors or deal with the relational semantics of the underlying data.

4.2.4 Choosing the platform

Web languages such as HTML5 and JavaScript have recently become popular in mobile app development for their portability across platforms. Browser-based mobile/Web platforms (e.g., Firefox OS, Chrome, and universal Windows apps) support this programming model by exposing high-level resource abstractions such as "contacts" and "photo gallery" to Web apps, as well as generic structured storage like IndexedDB; they are implemented in a customized, UI-less browser runtime, instead of app-level libraries. All resource accesses by apps are mediated by the browser runtime, although it only enforces all-or-nothing access control.

For our Earp prototype, we chose a browser-based platform, Firefox OS, allowing us to easily add fine-grained protection to many new and legacy APIs. Earp also retains coarse-grained protection on other legacy APIs (e.g., raw files), allowing us to demonstrate Earp's power and flexibility with substantial apps (Section 4.6).

It is possible to adapt Earp to a conventional mobile platform like Android. For storage, we could port SQLite into the kernel and add access-control enforcement to system calls; alternatively, we could create dedicated system services to mediate database accesses and enforce access-control policies. Non-cooperative apps would be confined by the reference monitor in either the kernel, or the services. For content providers, we could modify framework to support capability relationships, and require apps to provide unforgeable handles that are similar to subset descriptors when they access data in content providers. In Section 4.8, we show the design and implementation of a framework for Earp-style content providers.

4.3 Data storage and protection

UNIX stores byte streams in files protected by owner ID and permission bits and accessed via file descriptors. Earp stores structured data in relational databases protected by permission policies and accessed via *subset descriptors*. Because structured data is more complex than byte streams, Earp must provide more sophisticated protection mechanisms than what is needed for files. Before describing these mechanisms, we give a brief overview of the relational data model and how it's used in Earp.

4.3.1 Data model

Earp represents structured data using a relational model. The same relational API is used for storage and inter-app services (Section 4.4). The

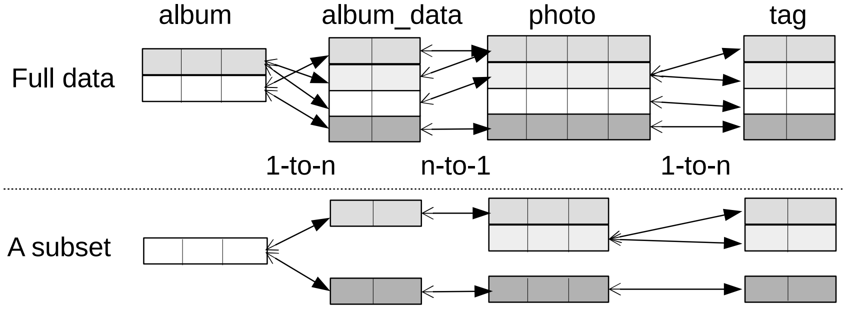


Figure 4.2: A relational representation of structured data. We show the entire data set and a subset chosen by a combination of row and column filtering. Relationships across tables are always bidirectional, but capability relationships are unidirectional as indicated by solid arrows.

back end of this API can be, respectively, a database or a service provided by another app.

Each data object in Earp is a *row* in some *table*, as shown in Figure 4.2. An object in one table can have relationships with objects in other tables. For example, a photo object is a row in the photo table with a column for raw image data, several columns for EXIF data (standard metadata such as the location where the photo was taken), and a relationship with the tag table, where tags store textual notes. Storing tags in a separate table allows photos to have an arbitrary number of tags that can be queried individually. Relationships in Earp are standard database relationships, as summarized below, but the concept of a capability relationship (Section 4.3.2) is a new contribution and the cornerstone of efficient access control in Earp.

Relationships have different cardinalities. For example, the relationship between a photo and its tags is *1-to-n* from the photo to its tags, or, equivalently, *n-to-1* from the tags to the photo. *1-to-1*, or, more precisely *(1|0)-to-1*, is a special case of n-to-1. For example, each digital camera has a single product profile which may or may not be present in the photo's EXIF.

Logically, the relationship between albums and photos is *n-to-n*, because a photo can be included in multiple albums and an album can contain multiple photos. Like many relational stores, Earp realizes n-to-n relationships by adding an intermediate table. In our example, we call the intermediate table *album_data*. The album-album_data relationship is 1-to-n, and the album_data-photo relationship is n-to-1. All four tables are illustrated in Figure 4.2.

4.3.2 Access rights

Access control lists. Each database in Earp is owned by a single app. Rows have very simple access control lists (ACLs) to control their visibility to other apps. Each row is either public, or private to a certain app. If a table does not have an `AppId` column, it can be directly accessed only by the owner of the database. If an Earp table has an `AppId` column, its value encodes the ACL: zero means that the row is public, positive n means that the row is private to the app whose ID is n. Any app can read or write public rows. Without an appropriate capability relationship (see below), apps can only read or write their own private rows.

Relationships create challenges for ACLs because they are traversed at run time and their transitive closure may include many objects. If ACLs were the only protection mechanism, an app that wants to share a photo with another app would have to modify the ACLs for all tags—either by making each ACL a list containing both apps, or by creating a group.

Capability relationships. A relationship is logically bidirectional. For example, given a photo, it is possible to retrieve its tags, and given a tag, it is possible to retrieve the photo to which it is attached. In Earp, however, only a single direction can confer access rights, as specified in the schema definition. These *capability relationships* are denoted as solid arrows in Figure 4.2.

We use $\mathtt{x}\overset{1:n}{\longleftarrow\!\!\!\blacktriangleright}\mathtt{y}$ to denote a 1-to-n capability relationship between tables `x` and `y`, which confers access rights when moving from the 1-side (`x`) to the n-side (`y`). Similarly, $\mathtt{x}\overset{n:1}{\longleftarrow\!\!\!\blacktriangleright}\mathtt{y}$ denotes an n-to-1 capability relationship that confers access when moving from the n-side to the 1-side. $\mathtt{x}\xleftrightarrow{n:1}\mathtt{y}$ denotes a non-capability relationship that does not confer access rights.

In the photo gallery example,

- $\mathtt{photo}\overset{1:n}{\longleftarrow\!\!\!\blacktriangleright}\mathtt{tag}$. Having a reference to a photo grants the holder the right to access all of that photo's tags, but not the other way around. Therefore, if an app asks for all photos with a certain tag, it will receive only the matching photos that are already accessible to it (via ownership, ACL, or capability relationship).

- $\mathtt{album}\overset{1:n}{\longleftarrow\!\!\!\blacktriangleright}\mathtt{album\_data}\overset{n:1}{\longleftarrow\!\!\!\blacktriangleright}\mathtt{photo}$. The intermediate table `album_data` realizes an n-to-n relationship with capability direction from `album` to `photo`.

Having access to an album thus confers access to the related objects in `album_data` and `photo`.

`album_data` and `tag` are both on the n-side of some `x`$\xrightarrow{1:n}$`y` relationship, and they are intended to be accessed only via capability relationships. For example, each tag is attached to a single photo and is useful only if the photo is accessible. Typically, such tables do not need ACLs.

We have not needed bidirectional capability relationships in Earp, and they would create cycles that make the access-control model confusing. Therefore, we decided not to support bidirectional capability relationships at the platform level. Earp prevents capabilities from forming cycles, ensuring that the transitive closure of all capability relationships is a directed acyclic graph (DAG).

Groups. A group can be created in Earp by defining a table with an appropriate schema. For example, to support albums that are shared by a group of apps, the app can define another table `album_access`, with `album_access`$\xrightarrow{n:1}$`album`. Each row in `album_access` is owned by one app and confers access to an album. With this table, even if an album is private to a certain app, it can be shared with other apps via entries in `album_access`.

Primary and foreign keys. Earp requires that all tables have immutable, non-reusable primary keys generated by the platform. The schema can also define additional keys. Therefore, the $(\mathit{database}, \mathit{table}, \mathit{primary\_key})$ tuple uniquely identifies a database row.

Cross-table relationships are represented via foreign keys in relational databases. A foreign key specifies an n-to-1 relationship: the table that contains the foreign key column is on the n-side, the referenced table is on the 1-side. If the foreign key column is declared with the `UNIQUE` constraint, the relationship is (1|0)-to-1.

Earp enforces that a foreign key references the primary key of another table and must guarantee *referential integrity* when the referenced row is deleted[3].

By default, when a referenced row is deleted, Earp sets the foreign keys of all referencing rows to `NULL`. However, when some referencing rows are no longer accessible or useful without the referenced row, the schema can explicitly prescribe that they should be deleted. For example, when a photo is deleted, its tags can be deleted because they are no longer accessible; it is also reasonable to delete rows referencing the photo in `album_data` because they no longer contain useful data.

4.3.3 App-defined access policies

ACLs and capability relationships are generic and enforced by Earp once the schema of a database or service is defined. To enable more expressive access control tailored for relational data, Earp also lets apps define schema-

[3] `https://www.sqlite.org/foreignkeys.html`

level *permission policies* on their databases and services. These policies govern other apps' access to the data.

A policy defines the following for each table:

1. AppID and default insert mode.

2. Permitted operations: insert, query, update, and/or delete.

3. A set of accessible columns (projection).

4. A set of columns with fixed values on insert/update.

5. A set of accessible rows (selected by a `WHERE` clause, in addition to ACL-based filtering).

The AppID is a number that identifies the controlling app as the basis for ACLs, much like the user ID identifies the user as the basis for interpreting file permission bits. The default insert mode indicates if data inserted into the database is public or private to the inserting app.

Data access in Earp is expressed by four SQL operations—insert, query, update, and delete—inspired by Android's SQLite API (omitting administrative functions like creating tables). Read-only access is realized by restricting the available SQL operations to query only. Control over writing is fine-grained: for example, an app can limit a client of the API to only insert into the database, without giving it the ability to modify existing entries.

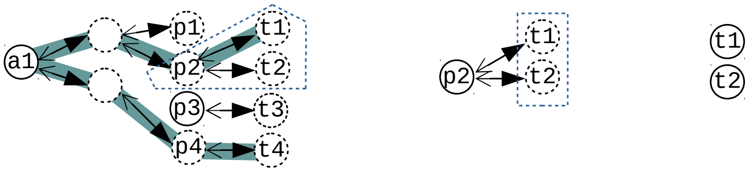
The permission policy can filter out certain rows (e.g., private photos) and columns (e.g., phone numbers of contacts), making them "invisible" to the client app. In addition, values of certain columns can be fixed on insert/update. For example, a Google Drive app can enforce that apps create files only in directories named by their official identifiers.

Just like the owner ID and permission bits of a file constrain the file descriptor obtained by a user when opening a file in UNIX, the permission policy constrains the subset descriptor (see below) obtained by a user when opening a database. While permission bits specify a policy for all users using coarse categories (owner, group, others), Earp lets apps specify initial permission policies for individual AppIDs, as well as the default policy. Figure 4.6 in Section 4.6 shows examples of policy definitions.

4.3.4 Subset descriptors

Apps in Earp access databases and services via subset descriptors. When an app opens a database or service that it owns, it obtains a full-privilege descriptor. If it opens another app's database or service, it obtains a descriptor with the owner's (default or per-app) permission policy.

Subset descriptors are created and maintained by Earp; apps manipulate opaque references to descriptors. Therefore, Earp initializes descriptors in accordance with the database owner's permission policy, and apps cannot tamper with the permissions of a descriptor (though descriptors can be downgraded, as discussed below).



d0: initial descriptor via opening the database (bold lines denote a join.)

d1: descriptor for a specific photo

d2: descriptor for the photo's tags

○ directly accessible entries ◌ indirectly accessible entries

```
var d0 = navigator.openDB('sys/gallery');
var cursor = d0.joinTransClosure(['album','album_data',
      'photo', 'tag'], where); // join
cursor.onsuccess = function(event) {
  ... // navigate to a row (the first bold line above)
  // d1: descriptor for photo in cursor's current row
  var d1 = cursor.getSelfDesc('photo');
  // d2: descriptor for the current photo's tags
  var d2 = cursor.getRefDesc('photo', 'tag');
}
```

Figure 4.3: A database join using an initial subset descriptor, then creating new descriptors to represent subsets of the result. The figure includes a visual depiction of the data accessible from the different descriptors.

Efficiently working with descriptors. An example of working with descriptors is shown in Figure 4.3. The app receives descriptor `d0` when it opens the database. It can use `d0` to access albums or photos as permitted by their ACLs. The code in Figure 4.3 will succeed in performing a join using `d0` because Earp verifies that all tables can be reached by traversing the capability relationships from a root table (`album` in this case), and that entries in different tables are related via corresponding foreign keys.

However, using `d0` is not always efficient for all tasks, because access rights on some objects can only be computed transitively. To minimize expensive cross-table checks, an app can create more descriptors that directly encode computed access rights over transitively accessible objects. Once such a descriptor is created, the app can use it to access the corresponding objects without recomputing access rights. In Figure 4.3, when the app successfully performs a query, join, or insert for a particular photo via `d0`, this proves to Earp that it can access the photo in question. Therefore, Earp lets it obtain a new descriptor `d2`, which allows the app to operate only on the entries in the `tag` table whose foreign key matches the photo's primary key. Access rights are verified and bound to `d2` upon its creation, thus subsequent operations on `d2` are not subject to cross-table checks. Any tag created using the `d2` descriptor will belong to the same photo because `d2` fixes the foreign key value to be the photo's primary key. As discussed in Section 4.3.5, the object graph library automates creation and management of descriptors.

The derived descriptor `d2` inherits the AppID, default insert mode, permitted operations, and accessible/fixed columns from `d0`. However, the set of accessible rows are recalculated to represent only the tags of a single photo, and a new fixed-column restriction is added for the foreign key, which must not be fixed previously.

Transferring and downgrading descriptors. An app can pass its descriptor to another app, a way to delegate access to the receiving app at run time. Transferring a descriptor generates a new copy of the descriptor in the receiving app. We say such a copy is derived from the original descriptor.

When delegating its access rights, an app may create a *downgraded* descriptor. For example, an app that has full access to an album may create a read-and-update descriptor for a single photo before passing it to a photo editor. A downgraded descriptor can also deny access to certain relationships by making the column containing the foreign key inaccessible.

Revoking descriptors. By default, a subset descriptor is valid until closed by the holding app. However, sometimes an app needs more control over a descriptor passed to another app. Therefore, Earp supports *transitive revocation*. When an app explicitly revokes a subset descriptor, all descriptors derived from it will also be revoked, including descriptors that are copied or transferred from it, as well as those generated based on query results. In this way, App A can temporarily grant access to App B by passing a descriptor `d` to it, then revoke App B's copy of `d` (and derived descriptors) afterwards by revoking the original copy in App A itself.

Creating relationships. A foreign key in Earp may imply access rights. For $\texttt{x}\xhookrightarrow{1:n}\texttt{y}$, foreign keys are never specified by the app. For example, inserting a tag for a photo can only be done via a descriptor generated for that photo's tags, i.e., `d2` in Figure 4.3, which fixes the foreign key value. This prevents an app from adding tags to a photo that it cannot access.

For $\texttt{x}\xhookrightarrow{n:1}\texttt{y}$, however, the app needs to provide a foreign key when creating a new row in `x`. For example, to add an existing photo to an album, the app needs to add a row in `album_data` with a foreign key referencing the photo. In this case, Earp must ensure that the app has some administrative rights over the referenced photo, because this operation makes the photo accessible to anyone that has access to the album. An analogy is changing file permissions in UNIX via `chmod`, which also requires administrative rights (matching UID or root).

To create such a reference, Earp requires an app to specify the foreign key value in the form of an unforgeable token. The app can obtain such a token via a successful insert or query on the referenced row, provided that the row is public or owned by the app. This proves that the app has administrative rights over the row.

4.3.5 Object graph library

As mentioned in Section 4.2, Earp provides a library that implements an object graph API on top of the relational data representation. Rows (e.g., photos) are represented as JavaScript objects. Related objects (e.g., photos and tags) are attached to each other via object references. The corresponding descriptors are computed and managed internally by the library. As Figure 4.1 illustrates for our running photo gallery example, an album can be retrieved (or stored) as a graph, and searching for photos with a certain tag can be done via a path query in this graph.

An app can use this library to conveniently construct a subgraph from an entry object that has capability or non-capability relationships with other objects. The lightweight nature of subset descriptors allows the library to proactively create descriptors as the app is performing queries. Internally, the library automates descriptor management and chooses appropriate descriptors for each operation. For example, it has dedicated descriptors for simple function APIs such as `addObjectRef` to create objects that have relationships with existing ones, as well as APIs that facilitate more complex operations, such as:

- `populateGraph`: populate a subgraph from a starting node (e.g., fetch all data from an album);

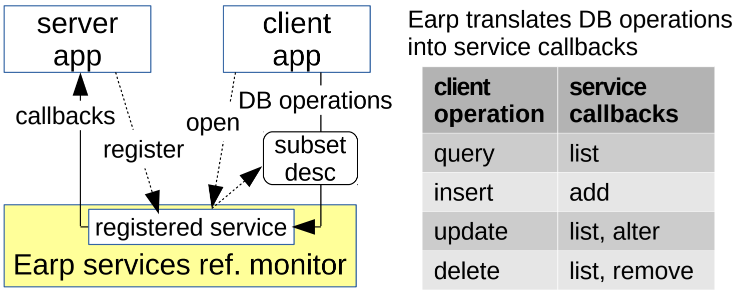
- `storeGraph`: store objects from a subgraph to multiple tables (e.g., store a new photo along with its tags);

- `queryPaths`: find paths in a subgraph that satisfy a predicate (e.g., find photos with a certain tag in an album).

4.4 Data sharing via inter-app services

In Earp, sharing non-persistent data between apps relies on the same relational abstractions as storage. In particular, data is accessed through subset descriptors that control which operations are available and which rows and columns are visible (just like for storage). The OS in Earp interposes on inter-app services, presents a relational view of the shared data, and is fully responsible for enforcing access control.

Figure 4.4 illustrates inter-app services in Earp. The *server app* is the provider of the data, the *client app* is a recipient of the data. In Earp, the server app defines and registers a named service, implemented with four *service callbacks*. To client apps, this service appears as a database with a set of *virtual*



Earp translates DB operations into service callbacks

client operation	service callbacks
query	list
insert	add
update	list, alter
delete	list, remove

Figure 4.4: Inter-app services in Earp.

tables and clients use subset descriptors to access this “database.” Defining virtual tables via callbacks is a standard idea, and a similar mechanism exists in SQLite[4]. Earp uses a subset of this interface tailored for the needs of mobile apps.

Virtual tables have the same relational model and are accessed through the same subset descriptors as conventional database tables (Section 4.3). The server app can define permission policies on virtual tables, in the same way as for storage databases. Like conventional tables, a virtual table can have a foreign key to another virtual table, defining a capability or non-capability relationship.

4.4.1 Implementing a relational service API

A service is implemented by defining four service callbacks: `list`, `add`, `alter`, and `remove`. The callbacks operate on *virtual tables* as follows.

[4] `https://www.sqlite.org/vtab.html`

- `list`: The server app provides a list of rows in the requested virtual table. This is the only set operation among the four callbacks. The server app also supplies values for the ACL column of any directly accessible table. Many use cases (Section 4.6), however, only rely on schema-level permission policies, so the server app may simply provide a dummy public value.

- `add`: Given a single row object, the server app adds it to the requested virtual table.

- `alter`: Given a single row object and new values for a set of columns, the server app updates that row in the requested virtual table.

- `remove`: Given a single row object, the server app deletes it from the requested virtual table.

Implementation of the service callbacks is necessarily app-specific. An app can retrieve data in response to a `list` invocation from an in-memory data structure, or fetch it on demand from a remote server via HTTP(S) requests. For example, `list` for the Google Drive service may involve fetching files, while `add` for the Facebook service may result in posting a status update.

4.4.2 Using a relational service API

Earp interposes on client apps' accesses to a service and converts standard database operations on virtual tables (query, insert, update, delete) into invocations of service callbacks. The reference monitor filters out inaccessible rows and columns and fixes column values according to the subset descriptor held by the client app.

- `query`: Earp invokes `list`, then filters the result set before returning to the client. Multi-table queries (joins) are converted to multiple `list` calls.

- `insert`: Earp sanitizes the client app's input row object by setting the values of fixed columns as specified in the descriptor, then passes the sanitized row to `add`.

- `update`: Earp invokes the `list` callback, performs filtering, sanitizes the new values, then invokes `alter` for each row in the filtered result set. This ensures that only the rows to which the client app has access will be updated, and that the client cannot modify columns that are inaccessible or whose values are fixed.

- `delete`: Earp invokes the `list` callback, performs filtering, then invokes `remove` for each row in the filtered result set.

4.4.3 Optimizing access-control checks

Earp's strategy of active interposition to enforce access control on inter-app services could reduce performance for certain server implementation patterns. We use several techniques to mitigate the performance impact on important use cases.

Separate data and metadata. Earp's filtering for `list` happens *after* the server app provides the data. Therefore, if the server returns a lot of unstructured “blob” data (e.g, raw image data associated with photos), possibly from a remote host, access control checks could be expensive.

In the common scenario where only metadata columns are used to define selection and access control criteria, the server app can greatly improve performance by separating the metadata and the blob data into two tables. The metadata table is directly visible to the client apps, and Earp performs filtering on it. The blob table is only accessible via a capability relationship (i.e., `metadata`$\xhookrightarrow{\text{n:1}}$`blob`). The client app receives the filtered result from the metadata table and can only fetch blobs that are referenced by the metadata rows.

Leverage indexing and query information. Although Earp does not require the server app to check the correctness or security of the data it returns in response to `list`, the server app can significantly reduce the amount of sent data if it already maintains indices on the data and takes advantage of the fact that Earp lets it see the actual client operation that invoked a particular callback.

For example, when a service exports a key/value interface, the server app can learn the requested key from Earp and return only the value for that key. Similarly, if the service acts as a proxy for a local database (e.g., a photo filter for the gallery), Earp sanitizes the client requests based on the client’s descriptor and passes the sanitized operations to the service. The service uses Earp’s database layer, which has a safe implementation of the relational model.

4.5 Implementation of Earp

We modified Firefox OS 2.1 to create the Earp prototype. The backend for storage is SQLite, a lightweight relational database that is already used by Firefox OS internally. Firefox OS supports inter-app communication based on a general message passing mechanism. It presents low-level APIs to send and receive JavaScript objects (similar to Android Binder IPC). Earp's inter-app service support is built on top of message passing, but presents higher-level APIs that facilitate access-control enforcement for structured data (similar to Android content providers which are built on top of Binder IPC). Our implementation of Earp consists of 7,785 lines of C++ code and 1,472 lines of JavaScript code (counted by CLOC[5]) added to the browser runtime and libraries.

4.5.1 Storing files

There are two ways to store files in Earp. When per-file metadata (e.g., photo EXIF data and ACLs) is needed, files can be co-located with the metadata in a database with file-type columns. Apps store large, unstructured blob data (e.g., PDF files) using file-type columns, and the only way for them to get handles to these files is by reading from such columns. This eliminates the need for a separate access-control mechanism for files. Internally, Earp stores the blob data in separate files and keeps references to these files in the database. This is a common practice for indexing files, used, for example, in

[5] `http://cloc.sourceforge.net/`

Android's photo manager and email client. Inserting a row containing files is atomic from the app's point of view. This allows Earp to consistently treat data and metadata, e.g., a photo and its EXIF.

If per-file metadata and access control are not needed, an app can store and manage raw files via directory handles. Access control is provided at directory granularity, and apps can have private or shared directories. Internally, Earp reuses the access-control mechanism for database rows to implement per-directory access control, simply by adding a directory-type column which stores directory references. The permissions on a directory are determined by the permissions on the corresponding database row.

4.5.2 Events and threads

JavaScript is highly asynchronous and relies heavily on events. Therefore, the API of Earp is asynchronous and apps get the results of their requests via callbacks.

Thread pool. Internally, all requests to storage and services are dispatched to a thread pool to avoid blocking the app's main thread for UI updates. The thread pool handles all I/O operations for database access and performs result filtering for inter-app services. After completing its processing of a request, Earp dispatches a success or error event to the main thread of the app, which invokes an appropriate callback.

A request may be processed by multiple concurrent threads to maximize

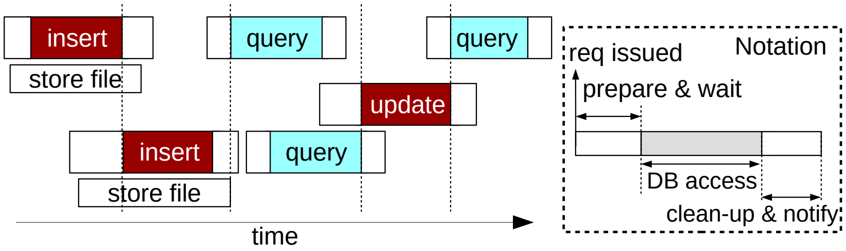


Figure 4.5: Constraints on request processing order in the thread pool.

parallelism. For example, inserting a row that contains n files will be processed by $n + 1$ threads, where the first n threads store the files and the last thread inserts metadata into the database. Although processed concurrently, such an insert request is atomic to apps, because they are not allowed to access the files until the insert finishes. If any thread fails, Earp aborts the operation and removes any written data.

Similarly, a request to a service can also be parallelized. For example, when processing an `update` request, Earp first uses a thread to invoke the `list` callback of the server app and to filter the result; for each row that passes the filter, Earp immediately dispatches an event to invoke the `alter` callback. If `alter` has high latency due to remote access, the server app can also parallelize its processing, e.g., by sending concurrent HTTP(S) requests.

Request ordering. When processing requests, Earp preserves the program order of all write requests (insert, update and delete) and guarantees that apps read (query) their writes. The critical section (database access) of a write waits for all previous requests to complete, while a read waits only for previous writes. Storing blob-type columns, as part of inserts or updates, is parallelized; however, a read must wait for the previous blob stores to complete. Note that an app could request an editable file or directory handle from a database query, but Earp does not enforce the order of reads and writes on the handle. It enforces the order when storing or replacing the whole blob using inserts or updates. Figure 4.5 shows an example of runtime request ordering.

4.5.3 Connections and transactions

A subset descriptor is backed by a database connection or a service connection. The program's order of requests is preserved *per connection*. When an app opens a database or a service, Earp creates a new connection for it. Descriptors that are derived from an existing descriptor inherit the same connection. However, the app can also request a new connection for an existing descriptor.

Earp exposes SQLite's support for transactions to apps. An app can group multiple requests in a transaction. If it does not explicitly use the API for transactions, each individual request is considered a transaction. Note that a transaction is for operations on a connection; requests on multiple descriptors could belong to a same transaction if they share the connection. The object graph library uses transactions across descriptors to implement the atomic version of `storeGraph`.

4.5.4 Safe SQL interface

SQL queries require `WHERE` clauses, but letting apps directly write raw clauses would create an SQL injection vulnerability. Earp uses structured objects to represent `WHERE` clauses and column-value pairs to avoid parsing strings provided by apps and relies on prepared statements to avoid SQL injection.

4.5.5 Reference monitor

The reference monitor mediates apps' access to data by creating appropriate descriptors for them and enforcing the restrictions encoded in the descriptor when processing apps' requests. Descriptors, requests, and tokens for foreign keys can only be created by the reference monitor; they cannot be forged by apps. They are implemented as native C++ classes with JavaScript bindings so that their internal representation is invisible to apps. These objects are managed by the reference counting and garbage collection mechanisms provided by Firefox OS.

App identity. An app (e.g., Facebook) often consists of local Web code, remote Web code from a trusted origin (e.g., `https://facebook.com`) specified in the app's manifest, and remote Web code from untrusted (e.g., advertising) origins. Earp adopts the app identity model from PowerGate [GJS15], and treats the app's local code and remote code from trusted origins as the same principal, "the app." Web code from other origins is considered untrusted and thus has no access to databases or services.

Policy management. Earp has a global registry of policies for databases and services, specified by their owners. Earp also has a trusted policy manager that can modify policies on any database or service.

4.6 Earp use cases

To illustrate how Earp supports sharing and access-control requirements of mobile apps, we implemented several essential apps based on Firefox OS native apps and utilities.

Photo gallery and editor. Gallery++ provides a user interface for organizing photos into albums and applying tags to photos (as in our running example). With the schema shown in Figure 4.2, Earp automates access control enforcement for Gallery++ and lets it define flexible policies for other apps. For example, when other apps open the photo database, they are granted access to their private photos and albums as well as public photos and albums, but certain fields like EXIF may be excluded.

Gallery++ can also share individual photos or entire albums with other apps (optionally including EXIF and tag information), by passing subset descriptors. For example, we ported a photo editing app called After Effects to Earp but blocked it from directly opening the photo database. Instead, this app can only accept descriptors from Gallery++ when the user explicitly invokes it for the photos she selected in Gallery++. When she finishes editing and returns from After Effects, Gallery++ revokes the descriptor to prevent further access.

Contacts manager. The Earp contacts manager provides an API identical to the Firefox OS contacts manager, thus legacy applications interacting with the manager all continue to work, yet their access is restricted according to the policies imposed by the Earp contacts manager.

The contacts manager stores contacts using seven tables: the main `contact` table in which the columns are simple attributes, five tables to manage attributes that allow multiple entries (e.g., `contact`$\xleftrightarrow{1:n}$`phone` and `contact`$\xleftrightarrow{1:n}$`email`), and the final table that holds contact categories with `category`$\xleftrightarrow{n:1}$`contact`. Categories can be used to restrict apps' access to groups of related contacts. Such a schema enables Earp-enforced custom policies, e.g., a LinkedIn app can be given access only to contacts in the "Work" category, without home address information.

Email. The Firefox OS built-in email client saves attachments to the world-readable device storage (SD card) when it invokes a viewing app to open the attachment.

The Earp email client allows attachments to be exported only to an authorized viewing app, which obtain a subset descriptor to the email app's database. The Earp email client also supports flexible queries from the viewing app, such as "show all pictures received in the past week," or "export all PDF attachments received two days ago".

Elgg social service and client apps. We use Elgg[6], an open-source social networking framework, to demonstrate Earp's support for controlled sharing of app-defined content. We customized Elgg to provide a Facebook-like social service where users can see posts from their friends. There are three components: the Elgg Web server, the Elgg local proxy app, and local client apps. Client apps are not authorized to directly contact the Elgg Web server. Instead, they must communicate with the Elgg local app which defines a service. This service acts as a local proxy and accesses remote resources hosted on the Web server.

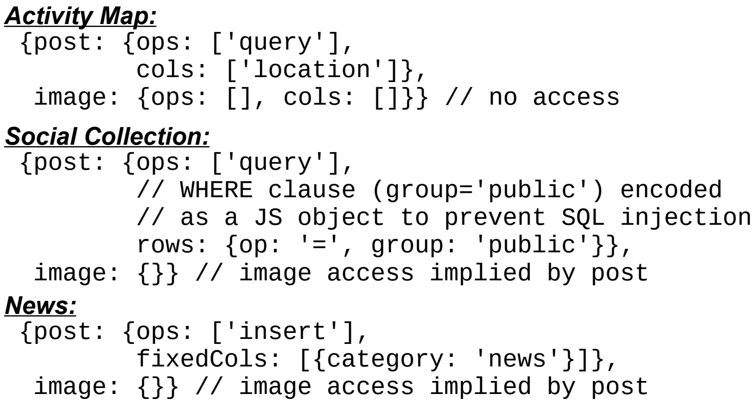
A post in Elgg is a text message with associated images. The Elgg app maintains two virtual tables, one for the post text (called `post`), the other for the images (called `image`), with a `post`$\xhookrightarrow{1:n}$`image` relationship.

The service callbacks use asynchronous HTTP requests to fetch data. To optimize bandwidth usage, images are only fetched when the requesting client app has access to the post with which they are associated.

Local access control in Earp provides a simple and secure alternative to OAuth. The Elgg local app defines policies for other apps based on user actions, e.g., via prompts. We implemented several client apps, and the policies for them are shown in Figure 4.6.

- An "activity map" app can read the `location` column in `post`, but not any textual or image data. The post-to-image capability relationship is

[6] `https://elgg.org/`



Activity Map:

```
 {post: {ops: ['query'],
         cols: ['location']},
  image: {ops: [], cols: []}} // no access
```

Social Collection:

```
 {post: {ops: ['query'],
         // WHERE clause (group='public') encoded
         // as a JS object to prevent SQL injection
         rows: {op: '=', group: 'public'}},
  image: {}} // image access implied by post
```

News:

```
 {post: {ops: ['insert'],
         fixedCols: [{category: 'news'}]},
  image: {}} // image access implied by post
```

Figure 4.6: Policies defined for Elgg client apps, represented as JavaScript objects.

unavailable to it, so it cannot fetch images even for accessible posts.

- A "social collection" app gathers events from different social networks. It can read all posts and associated images from the "public" group.

- A "news" app has insert-only access to the service, which is sufficient for sharing news on Elgg. The policy fixes the `category` column of any inserted post to be "news", preventing it from posting into other categories.

Google Drive and client apps. The Google Drive proxy app in Earp provides a local service that mediates other apps' access to cloud storage, avoiding the need for OAuth. Client apps enjoy the benefits of cloud storage without having to worry about provider-specific APIs or managing access credentials. The proxy app presents a collection of file objects containing metadata (folder and file name) and data (file contents) to other apps. It services requests from client apps by making corresponding HTTPS requests to Google's remote service. We have ported two client apps to use the service.

- DriveNote is a note-taking app which stores notes on the user's Google Drive account via the local proxy. The proxy allows it to read/write files only in a dedicated folder. Earp enforces this policy, ensuring that queries do not return files outside of this folder, and fixing the `folder` column on any update or insert operation.

- Gallery++ is a system utility, thus the Google Drive proxy app trusts it with access to all files. Gallery++ can scan and download all images stored on Google Drive.

4.7 Performance

We evaluate the performance of Earp on a Nexus 7 tablet, which has 2GB of DDR3L RAM and 1.5GHz quad-core Qualcomm Snapdragon S4 Pro CPU.

4.7.1 Microbenchmarks

We run various microbenchmarks to measure Earp's performance for storage and inter-app services. Figure 4.7 shows Earp's run time relative to Firefox OS.

DB-only workloads (contacts). We measure the time to insert new contacts, enumerate 500 contacts, and find a single contact matching a name or a phone number from the 500; the base line is the contacts manager in Firefox

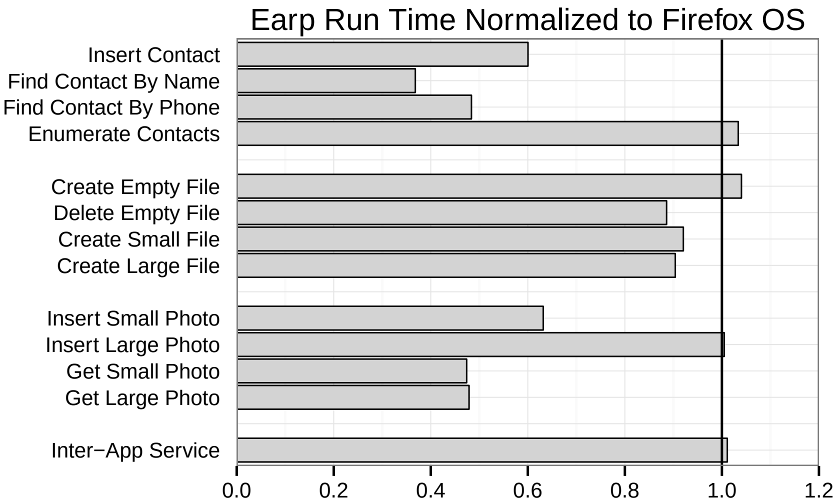


Figure 4.7: Microbenchmark results for storage and services. Smaller run time indicates better performance.

OS which uses IndexedDB. Earp outperforms the baseline for all workloads except enumerating contacts, where it is about only 3% slower.

Earp' performance is explained by its (1) directly using SQLite, while Firefox OS uses IndexedDB built on top of SQLite, (2) directly maping an object's fields into table columns, whereas IndexedDB uses expensive serialization to store the entire object, (3) using SQLite's built-in index support, whereas IndexedDB needs to create rows in an index table for all queryable fields of every object, (4) more complex data structure for contacts (six tables as opposed to a single serialized row for the baseline), which affords sophisticated access control but requires a bit more time to perform joins.

File-only workloads. We measure the time to create/delete empty files and write small (18KB)/large (3.4MB) files using Earp's directory API; the base line is Firefox OS' DeviceStorage API. Earp has comparable performance to the baseline, where the -11%∼4% difference in run time is due to different implementations of these APIs. Note that the measured times include event handling, e.g., dispatching to I/O threads and complete notification to the app.

DB-and-file workloads (photos). The measurements include inserting small, 18 KB, and large, 3.4 MB, photos with metadata, and retrieving them; the baseline is inserting/retrieving the same photo files and their metadata into the MediaDB library in Firefox OS, which uses IndexedDB. Earp largely outperforms the baseline, mostly because of the differences between SQLite and IndexedDB, as explained in the contacts experiments. When inserting large photos the run time is dominated by writing files so performance is very close (<1%) to the baseline.

Inter-app service. We measure the run time for retrieving 4,000 2 KB messages from a different app using Earp's inter-app service framework. The baseline uses Firefox OS' raw inter-app communication channel to implement an equivalent service, where requests are dispatched to Web worker threads (equivalent to Earp's thread pool). Figure 4.7 shows that Earp performs roughly the same as the baseline, and the time spent for access control (result filtering) is negligible.

	Baseline	Earp	slowdown
Elgg: read 50 posts	1623±102	1755± 99	8%
Elgg: upload 50 posts	5748±152	5888±117	2%
Google Drive: read 10 files	1310± 77	1392±120	6%
Google Drive: write 10 files	2828±217	2923±253	3%
Email: sync 200 emails	4725±433	4416±400	-6%

Table 4.1: Latency (msec) measured for macrobenchmarks on Earp applications.

4.7.2 Macrobenchmarks

Table 4.1 reports end-to-end latency for several real-world workloads described in Section 4.6.

Remote services. We measure the latency of client apps (Elgg client and DriveNote) accessing remote services (Elgg and Google Drive) by communicating with local proxy apps for these services. The baseline is the local proxy apps performing the same tasks by directly sending requests to their remote servers. The workloads include reading/uploading fifty posts with images via Elgg and reading/uploading ten 2KB text files via Google Drive. Table 4.1 shows that communicating with local proxy apps adds 3%∼8% latency, due to extra data serialization and event handling.

Email. We measure the latency of downloading 200 emails. The baseline is Firefox OS' email app which stores emails using IndexedDB. As shown in the "Email: sync" row of Table 4.1, Earp achieves similar performance storing the emails in an app-defined database.

4.8 Applicability to Android

Earp requires providing abstractions for structured data at the platform level, which is not satisfied by more popular platforms like Android and iOS (Section 2.1). However, this section discusses the feasibility of applying some of Earp's core concepts to Android, despite the differences in software architecture compared to Firefox OS. We demonstrate that Earp's model could be partially adopted in an incremental way for native platforms like Android.

Android's database support is provided by the SQLite library, which uses a single file to store each entire database. This makes it difficult to directly achieve fine-grained database access control across apps, because it would require porting SQLite into the kernel or a system service that implements access checks, which would be a significant change to the software architecture. We do not investigate this approach in the dissertation.

On the other hand, Android's content provider mechanism (Section 2.2) imposes a standard, database-like API for cross-app communications, with mostly coarse-grained access control. In this case, the app that implements the content provider (server app) owns all of the data, and is responsible for defining the access control requirements when sharing data with other apps. We can thus apply Earp's model for inter-app services to Android content providers.

Once content providers have structure-aware protection, we could create wrapper content providers for databases, which run in dedicated system

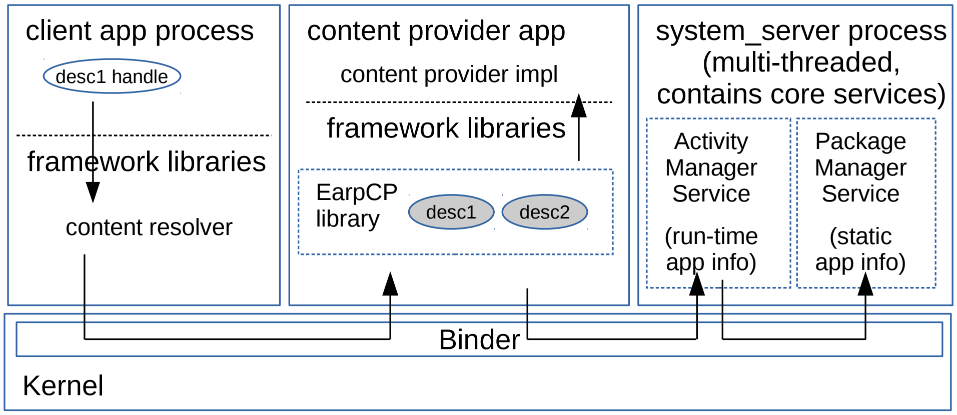


Figure 4.8: The EarpCP library added to Android's content provider framework. The states of subset descriptors are maintained by the library, and client apps only hold references (handles) to descriptors.

services, in order to achieve Earp-style protection for storage. Therefore, in principle, protecting content providers can be viewed as a general solution for both inter-app services and storage. However, using a content provider to mediate database accesses may add significant overhead due to the additional inter-process communication; also, content providers do not have a strict SQL interface, which would cause backward-compatibility issues.

4.8.1 Earp for Android content providers

As Figure 2.2 shows, access control is enforced by user-level framework libraries for content providers. Thus we cannot enforce Earp-style access control at the platform level. Instead, we choose to add a new framework library, EarpCP, to implement Earp-style data protection (Figure 4.8). Since the content provider app owns the data, we do not need to prevent it from bypassing its own access checks. This approach cannot prevent bugs in the owning app from compromising its own access control logic, but it does prevent client apps from bypassing the checks, and provides a common framework for structure-aware access control for content providers.

Data objects and relationships. A data object in a content provider is represented as a row that has several column values, similar to a row in relational databases. We say objects of the same type collectively form a *virtual table*. Suppose the previous photo gallery example (Figure 4.2) is implemented as a content provider, then albums, photos and tags can be represented as three different virtual tables. Note that virtual tables are not necessarily backed by an SQLite database; even if they are, there may not be a one-to-one mapping between virtual tables and database tables.

Content providers use different URIs to locate data objects. A URI either identifies a collection of objects (optionally with query parameters to filter the result), or a single object (typically with a unique ID as part of the URI). Normally, each virtual table has one standard format to identify a single row, with the ID as a path segment in the URI; for example, `content://gallery/photos/12` may represent the photo with `ID=12`. However, there are often multiple ways to identify a collection of rows for the same virtual table; for example `content://gallery/photos` and `content://gallery/albums/3/photos` may represent all photos and photos belonging to the album with `ID=3`, respectively. In this way, a cross-table query can be encoded as

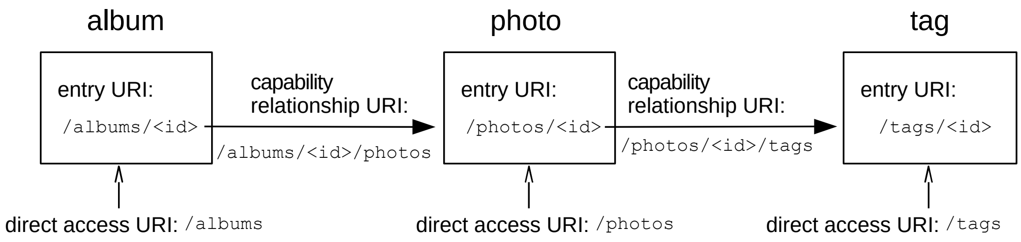


Figure 4.9: URIs and capability relationships in the photo gallery example. The common prefix of all URIs (`content://gallery`) is omitted.

a URI, which allows us to conveniently track relationships between different types of objects.

The EarpCP library requires the content provider to specify the information about virtual tables, including the URI formats for single objects or collections. Then the content provider needs to annotate cross-table relationships, where each relationship is specified as a rule to generate a cross-table URI based on an exiting object. The content provider also needs to specify whether each cross-table relationship confers access rights, i.e., whether it is a *capability relationship*. For example, the album-to-photo relationship can be specified by a rule that allows an album (e.g., `content://gallery/albums/3`) to confer access rights to its photos (e.g., `content://gallery/albums/3/photos`). All capability relationships should form a DAG. The gallery example is illustrated in Figure 4.9.

Unlike foreign keys in databases, the URI-based cross-table relationships only have one direction. The content provider could implement both album-to-photo and photo-to-album relationships with different URI formats, but only one of them can confer access rights, due to the DAG requirement.

Subset descriptors. In Android, a client app does not explicitly open a content provider to get a handle for the IPC connection. Instead, it needs to specify a content URI on every access, which is used by Android to locate the content provider. However, Earp should allow an app to use different descriptors to access the same content provider. The app thus needs to know the IDs (handles) of descriptors in order to differentiate them. We create a new API for opening, manipulating and closing descriptors, which is a standard URI path—when the client app queries this URI, it opens the content provider and gets a subset descriptor ID as the result; when it updates the URI with a descriptor ID specified in a query parameter, it downgrades this descriptor according to the new values provided in this update; when it deletes the URI with a specified descriptor ID, it closes this descriptor.

Subset descriptors encode access rights to the content provider. To prevent a client app from tampering with the encoded state, EarpCP is responsible for keeping the descriptor state in the content provider app. When a descriptor is created, it is assigned a large random number, which is used as the handle (descriptor ID) for the client app. The client app can transfer such a descriptor by simply sending the random number to another client app. Since the probability of correctly guessing a large random number is negligible, a client app cannot claim the possession of a descriptor unless it actually receives it.

Like in Earp, a subset descriptor in EarpCP encodes per-table restrictions including permitted operations, column filtering, fixed columns, and row filtering. Row filtering is achieved with two mechanisms: a set of directly accessible URIs and additional mandatory query parameters. For example, accessible URI `content://gallery/photos` with mandatory query parameter `ID=3` is equivalent to accessible URI `content://gallery/photos/3` without mandatory query parameters.

In order to use a descriptor to access data in the content provider, the client app must append the descriptor handle as a parameter to the URI. For each query or insert operation, if requested, EarpCP will create a new descriptor for each entry in the result that encodes access rights to just that entry, and append the handle in the result returned to the client app. This similar to Earp's mechanism of deriving descriptors as shown in Figure 4.3.

Per-app permission policies. The content provider needs to specify the access rights carried by the initial descriptor obtained upon open by each client app. Such initial access rights are the permission policy for the client app (Section 4.3.3).

4.8.2 Audio library in Media Provider

We use Android's Media Provider as an example to demonstrate the usage of EarpCP. It manages the user's image, audio and video libraries. Since the image and video libraries have similar and simpler data structures com-

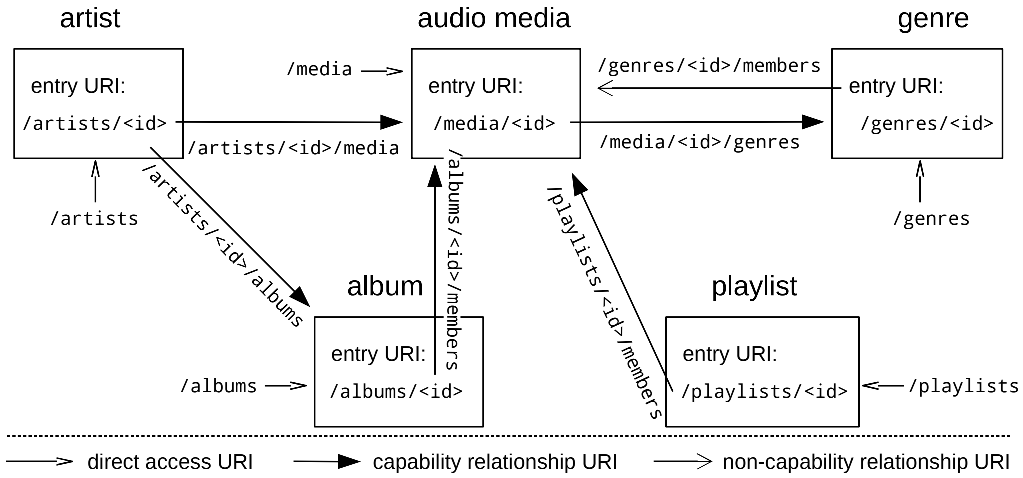


Figure 4.10: URIs and capability/non-capability relationships in the audio library of Media Provider. The common prefix of all URIs (`content://external/media/audio`) is omitted.

pared to Earp's photo gallery, we focus on the audio library, which has richer structures including artists, albums, playlists and genres.

We ported Android's audio library to use EarpCP, with minor modifications such as adding artist-to-audio and album-to-audio URIs, which are alternatives to directly using audio URIs with query parameters. The new URIs make it more convenient for us to implement capability relationships. The data model is shown in Figure 4.10. There are five virtual tables, connected with five capability relationships and one non-capability relationship (genre-to-audio). The capability relationships enable collection sharing based on artists, albums or playlists, which transitively grant access to member audio files. The non-capability relationship allows query by genres, but the result will include only the audio files that are already accessible.

	w/o EarpCP	w/ EarpCP	slowdown
create audio library	950±31	972±29	2%
browsing audio library	439±30	471±27	7%

Table 4.2: Time (msec) measured for running benchmark workload with and without EarpCP.

Passing a descriptor for an artist to another app will transitively grant the receiving app access to the albums and audio files of this artist. These objects need different URIs to locate, all of which are allowed by the capability relationships. In comparison, Android's URI permission mechanism only allows passing capability over a particular URI.

Performance. We measure the performance overhead added by EarpCP for the audio library, on the same Nexus 7 tablet as used in Section 4.7. We use a benchmarking client app that performs read/write operations. The workload includes 1) creating an audio library with metatata for 200 fake audio files, which are organized in 20 albums from 20 artists, and 2) retrieving the entire audio library created in the previous step by browsing through all artists, albums and audio files.

Table 4.2 shows the comparison of running the workload with and without EarpCP protection. EarpCP adds 2% overhead for creating the audio library and 7% for browsing. These overheads are low because EarpCP does not need extra IPC round trips to communicate the subset descriptors. Instead, descriptor handles are piggybacked on regular URIs sent to the content provider and results returned to the client app.

Chapter 5

Related work

Related work of this dissertation includes systems that use information flow techniques to confine untrusted code, as well as efforts to make access control fine-grained and more flexible in mobile platforms. We also discuss systems that use related techniques in scenarios other than mobile platforms.

5.1 Information flow

Maxoid's confinement model is a form of information flow control that keeps data secure when processed by untrusted code. In fact, invoking untrusted code on sensitive data is a classic security problem that has been addressed by several desktop/server systems, such as language-level decentralized information flow control (DIFC) [ML97, CF07, LGV+09, AGL+12], OS-level DIFC [KYB+07, EKV+05, ZBWKM06, JAF+], PL-OS DIFC [RPB+09] and architectural-OS information flow [TOL+11]. These systems assign security labels to sensitive data, and control how labeled data can be propagated across different domains. For mobile platforms specifically, a DIFC-style system for Android is proposed in [JAF+]. Like other DIFC systems, it is not transparent to untrusted legacy apps, since they need to be written in a way that explicitly obey the DIFC rules.

TaintDroid [EGC$^+$10] is a fine-grained taint tracking system for Android which detects data leakage. In order to achieve fine granularity, it modifies different layers in Android's software stack, including the Dalvik VM. However, the modified Dalvik VM has a security limitation, which is that it does not detect implicit data leaks through control flows. For example, a branching statement can leak one bit of information about the branch condition, but TaintDroid does not track such leakage. A malicious app could accumulate information via a large number of branches, constructing a high-bandwidth channel. In contrast, Maxoid is more conservative, i.e., a delegate's (even in native code) output is always controlled.

There are also systems built on top of TaintDroid. AppFence [HHJ$^+$11] uses TaintDroid to stop apps from sending the user's sensitive data over the network. Like TaintDroid, it identifies system-wide sensitive data such as the device ID and contacts as the source of sensitive data, rather than per-app private data as in Maxoid. AppFence uses some heuristic techniques to reduce disruptions to legacy apps, such as providing a fake value as the device ID sent over the network. It does not control intra-device taint propagation as Maxoid does, which would be a more severe problem if per-app private data were also tracked and protected. CleanOS [TAB$^+$12] uses TaintDroid to track the propagation of an app's sensitive data and protects them via encryption. Unlike Android, its focus is reducing the lifetime of an app's clear-text sensitive data on the device.

5.2 Flexible access control

A number of recent systems have been proposed to change the coarse-grained, ineffective access control in existing mobile platforms. Before Android 6.0, apps could only request permissions at install time, but there have been research systems [OMEM12, NKZ10, CNC10, BRSS11, XSA12, BGH+12] that enable flexible runtime permission granting or revoking.

SE Android [SC13] and FlaskDroid [BHS13] add mandatory access control to Android. Systems like ServiceOS [MWL13], Bubbles [TMO+12], IPC Inspection [FWM+11] and QUIRE [DSP+11] provide apps different access rights when they execute in different contexts. While these systems improve security by enforcing stricter access control, they do not provide information flow control like Maxoid.

FlaskDroid also provides a design pattern for content providers to filter query results for different client apps, which is based on SQL views. However, unlike Earp, it is limited to SQLite-based content providers and does not consider capability relationships across table. By contrast, Earp supports all types of inter-app services, including proxies for remote servers.

Pebbles [SBL+14] is another TaintDroid-based system, which is closely related to Earp. It modifies Android's SQLite and XML libraries and uses TaintDroid to discover app-level data structures across different types of storage, at the cost of considerable overhead in certain workloads. Pebbles relies on developers using certain design patterns consistently to infer the structure of data and it is implemented in app-level libraries. In contrast, Earp explicitly requires apps to expose their structures to the platform, and its enforcement is part of the platform.

While Pebbles and Earp make data objects fine-grained for protection purpose, there are also systems that make the security principals fine-grained, which is a practice of privilege separation. AdSplit [SDW12] and AdDroid [PFNW12] split an app and its untrusted advertising into separate processes, while FlexDroid [SKC$^+$16] achieves privilege separation for third-party libraries using hardware-based fault isolation. These techniques are orthogonal to Maxoid and Earp, and similar approaches can enable Maxoid delegates to use advertising in a separate process without network disruption.

πBox [LWG$^+$13] is an Android-based platform that has an extend sandbox that spans the user's device and the cloud. It can confine untrusted apps with network access limited to a trusted cloud. It thus requires that all apps' servers must be deployed on the trusted cloud. Similar approaches might help address the limitation that Maxoid must block network for delegates. However, the required trusted cloud does not exist for commodity mobile platforms.

5.3 Related techniques for other platforms

Maxoid and Earp have also been influenced by some other systems that are not designed for mobile platforms. We discuss related techniques used in these systems.

Using different views of data for security. Solitude [JSDG08], Apiary [PN10] and Mbox [KZ13] use union file systems for application fault containment or sandboxing in Linux. The design of file system support in Maxoid is inspired by these systems, but our goals and approaches are suited for mobile apps' collaboration on sensitive data.

Fine-grained protection in databases. Earp heavily relies on relational databases. Traditional access control systems for relational databases [BL08, BJS96, FSG+01, OSM00, J+09, RMSR04, GB14] are based on users or roles with relatively static policies. More recently, IFDB [SL13] showed how DIFC can be integrated with a relational database. Like Earp, IFDB also discusses foreign key issues, but focuses on potential information leakage due to referential integrity enforcement. Earp's capability relationships focus on a different problem, and they are used to express protection requirements for mobile apps' inter-related data objects.

Protection on Web platforms. Earp's prototype is a browser-based platform. For JavaScript code running in Web browsers, there are several systems that enable flexible policies [CGZ10, ML10, JDRC10], controlled object sharing [MFM10, PDL+11], or confinement [SYM+14, HPD13, IW12]. While all these systems improve access control enforcement, they do not provide platform-level abstractions for relational data.

Downgradable and transferable access rights. Google has proposed a distributed authorization system, Macaroons [BPÃE$^+$14], which allows access rights to be delegated across protection domains, and access rights can be downgraded by further restricting the set of accessible objects and the possible contexts where they can be used. DCAC [XDH$^+$14] uses hierarchically-named attributes to represent access rights held by processes, where an attribute can be downgraded to its child attributes and delegated to other processes. While Earp's subset descriptors share similarities with these mechanisms, the relational model and capability relationships can enable more expressive policies.

Native relational stores. Like Earp, there are previous efforts to make relational data directly supported by the OS, notably Microsoft's cancelled project WinFS[1]. WinFS contains a database engine to natively support SQL, and files/directories are implemented on top of the database. While WinFS had fine-grained access control, it was still based on per-object permissions; WinFS was developed before mobile platforms become popular, and traditional desktop apps that rely on files suffered performance penalties due to database-managed metadata. Earp's database-centric approach suits the current practice of mobile development where databases are the de facto hub for storage [SBL$^+$14]. Finally, because Earp uses an unmodified file system for raw files (unlike WinFS), it provides compatibility file APIs that have no performance overhead.

[1] `https://msdn.microsoft.com/en-us/library/Aa479870.aspx`

Chapter 6

Conclusion

Security mechanisms provided by existing mobile platforms often fail to support requirements of modern apps that frequently interact with each other. First, existing platforms do not provide information flow control for inter-app data exchanges, resulting in data leaks in many common scenarios; we present Maxoid, a system that uses custom views of state to make conservative information flow confinement transparent, achieving much stronger secrecy and integrity guarantees with backward compatibility. Second, modern apps heavily rely on structured data with inter-related objects, but existing platforms' security mechanisms are blind to such structures and relationships, causing unprincipled, ad hoc data sharing and protection practices; we present Earp, a new mobile platform which protects structured data—the abstraction used by apps—at the platform level, enabling principled data protection and sharing.

Bibliography

[AGL$^+$12] Owen Arden, Michael D George, Jed Liu, K Vikram, Aslan Askarov, and Andrew C Myers. Sharing mobile code securely with information flow control. In *IEEE Symposium on Security and Privacy*, 2012.

[BGH$^+$12] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. AppGuard – real-time policy enforcement for third-party applications. Technical Report A/02/2012, MPI-SWS, 2012.

[BHS13] Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi. Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. In *USENIX Security Symposium*, 2013.

[BJS96] Elisa Bertino, Sushil Jajodia, and Pierangela Samarati. Supporting multiple access control policies in database systems. In *IEEE Symposium on Security and Privacy*, 1996.

[BL08] Ji-Won Byun and Ninghui Li. Purpose based access control for privacy protection in relational database systems. *The VLDB Journal*, 17(4):603–619, 2008.

[BPÃE$^+$14] Arnar Birgisson, Joe Gibbs Politz, ÃŽlfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[BRSS11] Alastair R. Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. MockDroid: trading privacy for application functionality on smartphones. In *International Workshop on Mobile Computing Systems and Applications (HotMobile).* ACM, 2011.

[CF07] Deepak Chandra and Michael Franz. Fine-grained information flow analysis and enforcement in a Java virtual machine. In *Annual Computer Security Applications Conference (ACSAC)*, 2007.

[CGZ10] Ramesh Chandra, Priya Gupta, and Nickolai Zeldovich. Separating web applications from user data storage with BSTORE. In *USENIX Conference on Web Application Development (WebApps)*, 2010.

[CNC10] Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. CRePE: Context-related policy enforcement for Android. In *Information Security Conference (ISC)*, 2010.

[DSP$^+$11] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S Wallach. QUIRE: Lightweight Provenance for Smart

Phone Operating Systems. In *USENIX Security Symposium*, 2011.

[EGC$^+$10] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. Taint-Droid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[EKV$^+$05] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazieres, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *ACM Symposium on Operating System Principles (SOSP)*, 2005.

[EOMC11] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of Android application security. In *USENIX Security Symposium*, 2011.

[FSG$^+$01] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed nist standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):224–274, 2001.

[FWM$^+$11] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, 2011.

[GB14] Marco Guarnieri and David Basin. Optimal security-aware query processing. In *International Conference on Very Large Data Bases (VLDB)*, 2014.

[GJS14] Martin Georgiev, Suman Jana, and Vitaly Shmatikov. Breaking and fixing origin-based access control in hybrid web/mobile application frameworks. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[GJS15] Martin Georgiev, Suman Jana, and Vitaly Shmatikov. Rethinking security of Web-based system applications. In *International World Wide Web Conference (WWW)*, 2015.

[HHJ$^+$11] 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 *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[HPD13] Jon Howell, Bryan Parno, and John R Douceur. Embassies: radically refactoring the Web. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[IW12] Lon Ingram and Michael Walfish. Treehouse: JavaScript sandboxes to help web developers help themselves. In *USENIX Annual Technical Conference*, pages 153–164, 2012.

[J$^+$09] Sumit Jeloka et al. *Oracle Label Security Administrator's Guide.* Oracle Corporation, release 2 (11.2) edition, 2009.

[JAF$^+$] Limin Jia, Jassim Aljuraidan, Elli Fragkaki, Lujo Bauer, Michael Stroucken, Kazuhide Fukushima, Shinsaku Kiyomoto, and Yutaka Miyake. Run-time enforcement of information-flow properties on Android. In *ESORICS 2013*.

[JDRC10] Karthick Jayaraman, Wenliang Du, Balamurugan Rajagopalan, and Steve J Chapin. Escudo: A fine-grained protection model for web browsers. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2010.

[JSDG08] Shvetank Jain, Fareha Shafique, Vladan Djeric, and Ashvin Goel. Application-level isolation and recovery with Solitude. In *ACM European Conference in Computer Systems (EuroSys)*, 2008.

[KYB$^+$07] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *ACM Symposium on Operating System Principles (SOSP)*, 2007.

[KZ13] Taesoo Kim and Nickolai Zeldovich. Practical and Effective Sandboxing for Non-root Users. In *USENIX Annual Technical Conference (ATC)*, 2013.

[LGV+09] Jed Liu, Michael D George, Krishnaprasad Vikram, Xin Qi, Lucas Waye, and Andrew C Myers. Fabric: A platform for secure distributed computation and storage. In *ACM Symposium on Operating System Principles (SOSP)*, 2009.

[LWG+13] Sangmin Lee, Edmund L. Wong, Deepak Goel, Mike Dahlin, and Vitaly Shmatikov. πbox: a platform for privacy-preserving apps. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[LZX+14] Tongxin Li, Xiaoyong Zhou, Luyi Xing, Yeonjoon Lee, Muhammad Naveed, XiaoFeng Wang, and Xinhui Han. Mayhem in the push clouds: Understanding and mitigating security hazards in mobile push-messaging services. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[MFM10] Leo A. Meyerovich, Adrienne Porter Felt, and Mark S. Miller. Object views: Fine-grained sharing in browsers. In *International World Wide Web Conference (WWW)*, 2010.

[ML97] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *ACM Symposium on Operating System Principles (SOSP)*, pages 129–142, October 1997.

[ML10] Leo A. Meyerovich and Benjamin Livshits. Conscript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *IEEE Symposium on Security and Privacy*, 2010.

[MWL13] Alexander Moshchuk, Helen J. Wang, and Yunxin Liu. Content-based isolation: rethinking isolation policy design on client systems. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[NKZ10] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: extending Android permission model and enforcement with user-defined runtime constraints. In *ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*. ACM, 2010.

[OMEM12] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically rich application-centric security in Android. *Security and Communication Networks*, 5(6):658–673, 2012.

[OSM00] Sylvia Osborn, Ravi Sandhu, and Qamar Munawer. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(2):85–106, 2000.

[PDL$^+$11] Kailas Patil, Xinshu Dong, Xiaolei Li, Zhenkai Liang, and Xuxian Jiang. Towards fine-grained access control in JavaScript contexts. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2011.

[PFNW12] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. Addroid: Privilege separation for applications and advertisers in Android. In *ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*, 2012.

[PN10] Shaya Potter and Jason Nieh. Apiary: Easy-to-use desktop application fault containment on commodity operating systems. In *USENIX Annual Technical Conference (ATC)*, 2010.

[RMSR04] Shariq Rizvi, Alberto Mendelzon, Sundararajarao Sudarshan, and Prasan Roy. Extending query rewriting techniques for fine-grained access control. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2004.

[RPB$^+$09] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical fine-grained decentralized information flow control. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2009.

[RT74] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Communications of the ACM (CACM)*, 17(7), 1974.

[Sal74] Jerome H. Saltzer. Protection and the control of information sharing in multics. *Communications of the ACM (CACM)*, 17(7), 1974.

[SB12] San-Tsai Sun and Konstantin Beznosov. The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Sytems. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[SBL$^+$14] Riley Spahn, Jonathan Bell, Michael Z. Lee, Sravan Bhamidipati, Roxana Geambasu, and Gail Kaiser. Pebbles: Fine-Grained Data Management Abstractions for Modern Operating Systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[SC13] Stephen Smalley and Robert Craig. Security enhanced (SE) android: Bringing flexible mac to Android. In *Network and Distributed System Security Symposium (NDSS)*, 2013.

[SDW12] Shashi Shekhar, Michael Dietz, and Dan S Wallach. AdSplit: Separating smartphone advertising from applications. In *USENIX Security Symposium*, 2012.

[SH14] Hossain Shahriar and Hisham M. Haddad. Content provider leakage vulnerability detection in Android applications. In *International Conference on Security of Information and Networks (SIN)*, 2014.

[SKC$^+$16] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Taesoo Kim, and Insik Shin. FlexDroid: Enforcing in-app privilege separation in Android. 2016.

[SL13] David Schultz and Barbara Liskov. IFDB: decentralized information flow control for databases. In *ACM European Conference in Computer Systems (EuroSys)*, 2013.

[SYM$^+$14] Deian Stefan, Edward Z Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazieres. Protecting users by confining JavaScript with COWL. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[TAB$^+$12] Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda. CleanOS: Limiting mobile data exposure with idle eviction. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[TMO$^+$12] Mohit Tiwari, Prashanth Mohan, Andrew Osheroff, Hilfi Alkaff, Elaine Shi, Eric Love, Dawn Song, and Krste Asanović. Context-centric security. In *USENIX Workshop on Hot Topics in Security (HotSec)*, 2012.

[TOL$^+$11] Mohit Tiwari, Jason Oberg, Xun Li, Jonathan K Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T Chong, and Timothy Sherwood. Crafting a usable microkernel, processor, and i/o system with strict and provable information flow security. In *International Symposium on Computer Architecture (ISCA)*, 2011.

[VGN14] Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of Google Play. In *ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2014.

[WXWC13] Rui Wang, Luyi Xing, XiaoFeng Wang, and Shuo Chen. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[XBL+15] Luyi Xing, Xiaolong Bai, Tongxin Li, XiaoFeng Wang, Kai Chen, Xiaojing Liao, Shi-Min Hu, and Xinhui Han. Cracking app isolation on Apple: Unauthorized cross-app resource access on MAC OS X and iOS. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[XDH+14] Yuanzhong Xu, Alan M. Dunn, Owen S. Hofmann, Michael Z. Lee, Syed Akbar Mehdi, and Emmett Witchel. Application-defined decentralized access control. In *USENIX Annual Technical Conference (ATC)*, 2014.

[XHK$^+$16] Yuanzhong Xu, Tyler Hunt, Youngjin Kwon, Martin Georgiev, Vitaly Shmatikov, and Emmett Witchel. Earp: Principled storage, sharing, and protection for mobile apps. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[XSA12] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: Practical policy enforcement for Android applications. In *USENIX Security Symposium*, 2012.

[XW15] Yuanzhong Xu and Emmett Witchel. Maxoid: Transparently Confining Mobile Applications with Custom Views of State. In *ACM European Conference in Computer Systems (EuroSys)*, 2015.

[ZBWKM06] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[ZJ13] Yajin Zhou and Xuxian Jiang. Detecting passive content leaks and pollution in Android applications. In *Network and Distributed System Security Symposium (NDSS)*, 2013.

Vita

Yuanzhong Xu was born in Hebei, China. He graduated from Shanghai Jiao Tong University in 2011 with a B.E. degree in Information Engineering. In August 2011, he entered the doctoral program in the Department of Computer Science at the University of Texas at Austin, where he received an M.S. degree in Computer Science in December 2015.

Permanent address: yxu@cs.utexas.edu

This dissertation was typeset with LaTeX[†] by the author.

[†] LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.