Earp: Principled Storage, Sharing, and Protection for Mobile Apps

Yuanzhong Xu Tyler Hunt Youngjin Kwon Martin Georgiev
Vitaly Shmatikov[†] Emmett Witchel

The University of Texas at Austin [†]*Cornell Tech*

Abstract

Modern mobile apps need to store and share structured data, but the coarse-grained access-control mechanisms in existing mobile operating systems are inadequate to help apps express and enforce their protection requirements.

We design, implement, and evaluate a prototype of Earp, a new mobile platform that uses the relational model as the unified OS-level abstraction for both storage and inter-app services. Earp provides apps with structure-aware, OS-enforced access control, bringing order and protection to the Wild West of mobile data management.

1 Introduction

Modern mobile apps communicate and exchange data with other apps almost as much as they communicate and exchange data 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.

Traditionally, operating systems have provided abstractions and protection for storing and sharing data. The data model in UNIX [34] is byte streams, stored in files protected by owner ID and permission bits and accessed via file descriptors. UNIX has a uniform access-control model for both 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 [40]. 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 (see §2).

In this paper, 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.

Our contributions. 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 [36] 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. Apps access data and system resources via the trusted browser runtime, which acts as the OS from the app's viewpoint. 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.

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—for example, elide sensitive data fields, support private photos and albums, filter contacts based on categories, or temporarily grant access to an attachment file. 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.

2 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—for example, a photo gallery is a collection of photos, each of which is tagged with user's notes.

Inadequate protection for storage. Existing platforms do not provide adequate support for mobile apps' data management. Without system abstractions for storing and protecting data, 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 [54] 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 [27, 44, 48, 49, 51]. 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 [36].

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. Android's reference monitor enforces only coarse-grained access control for content providers based on static permissions specified in app manifests [2]. Even though permissions can be specified for particular URI paths, they can only be used for static, coarse categories (e.g., images or audio in Media Content Provider) because it is impossible to assign different permissions to dynamically created objects, nor enforce custom policies for different client apps. 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 [38, 54].

Android also has a URI permission mechanism [1] for fine-grained, temporary access granting. 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.

In iOS, apps cannot directly share data via the file system or background services. For example, to share a photo, apps either copy it to the system's gallery, or use app extensions [24] which require user involvement (e.g., using a file picker) for every operation.

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 [18, 19]. Management of OAuth tokens is notoriously difficult and many apps badly mishandle them [48], 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 [44]. In 2015, a bug in Facebook's OAuth protocol allowed third-party apps to access users' private photos stored on Facebook's servers [14].

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.

3 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 paper, we describe how this is achieved in Earp.

3.1 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* (§5), 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. Earp uses files and directories internally, thus avoiding the historical performance problems of implementing a file system on top of a database [50].

3.2 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 (§2).

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.

3.3 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 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 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.

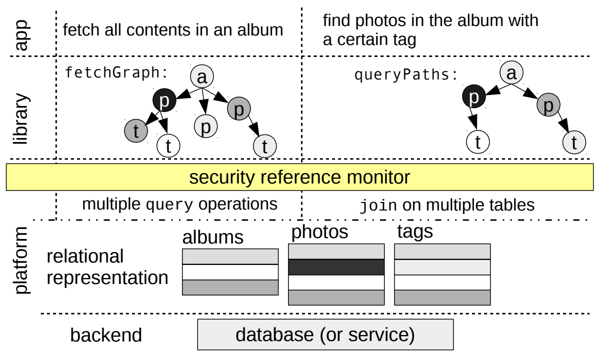

app
fetch all contents in an album
find photos in the album with a certain tag
library
fetchGraph:
queryPaths:
a
p
t
security reference monitor
multiple query operations
join on multiple tables
platform
relational representation
albums
photos
tags
backend
database (or service)

Figure 1: Platform- and library-level representations of structured data in Earp.

3.4 Choosing the platform

Mobile apps are often written in portable Web languages such as HTML5 and JavaScript [46, 47]. 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 (§7.1).

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 the reference monitor to support capability relationships, and require apps to provide unforgeable handles that are similar to subset descriptors when they access data in content providers.

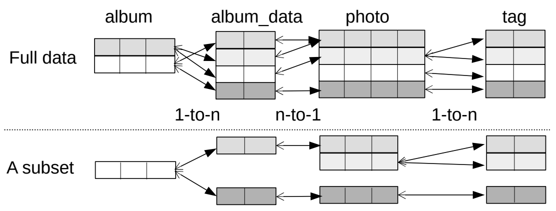


Figure 2: A relational representation of structured data. We show the entire data set and a subset chosen by a combination of row and column filtering. Relationships across tables are always bidirectional, but capability relationships are unidirectional as indicated by solid arrows.

4 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.

4.1 Data model

Earp represents structured data using a relational model. The same relational API is used for storage and inter-app services (§5). The 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 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 (§4.2) 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 2.

4.2 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 2.

We use $\mathtt{x}\xleftrightarrow{1:n}\mathtt{y}$ 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, $\mathtt{x}\xleftrightarrow{n:1}\mathtt{y}$ 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,

- $\mathtt{photo}\xleftrightarrow{1:n}\mathtt{tag}$. 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).
- $\mathtt{album}\xleftrightarrow{1:n}\mathtt{album\_data}\xleftrightarrow{n:1}\mathtt{photo}$. 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`$\xleftrightarrow{1:n}$`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`$\xleftrightarrow{n:1}$`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 $(database, table, primary\_key)$ 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 [41].

For `x`$\xleftrightarrow{1:n}$`y` where `y` does not have ACLs, when the referenced row (e.g., a photo) is deleted, the referencing rows (e.g., tags) will be deleted as well, because they are inaccessible and the deleting app has the (transitive) right to delete them.

For other types of relationships, when the referenced row (e.g., a photo) is deleted, Earp by default sets the foreign keys of the referencing rows (e.g., rows in `album_data`) to `NULL`. If these rows no longer contain useful data without the foreign key, the schema can explicitly prescribe that they should be deleted. For `album_data`, it is reasonable to delete the rows because they are merely intermediate relations between albums and photos.

4.3 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-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.

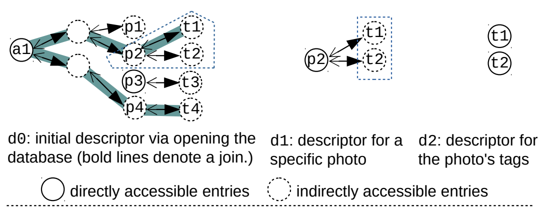
Data access in Earp is expressed by four SQL operations—insert, query, update, and delete—inspired by Android's SQLite API (omitting administrative functions like creating tables). Read-only access is realized by restricting the available SQL operations to query only. Control over writing is fine-grained: 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 6 in Section 7.1 shows examples of policy definitions.

4.4 Data-access APIs

Earp provides two levels of APIs to access relational data: direct access via subset descriptors and object-graph access via a library.


a1
p1
p2
p3
p4
t1
t2
t3
t4
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 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.

4.4.1 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).

Efficiently working with descriptors. An example of working with descriptors is shown in Figure 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 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 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 §4.4.2, the object graph library automates creation and management of descriptors.

Transferring and downgrading descriptors. An app can pass its descriptor to another app or it can create a new descriptor based on the one it currently holds (e.g., create `d1` based on `d0` in Figure 3). When a new descriptor is generated based on an existing one, all access restrictions are inherited. For example, if the existing descriptor does not include some columns, the new one will not have those columns, either; if the existing descriptor is query-only, so will be the new one; fixed values for columns, if any, are inherited, too.

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[1] 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 $\mathtt{x} \xleftrightarrow{1:n} \mathtt{y}$, 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 3, which fixes the foreign key value. This prevents an app from adding tags to a photo that it cannot access.

For $\mathtt{x} \xleftrightarrow{n:1} \mathtt{y}$, 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

[1]Transferring a descriptor generates a new copy of the descriptor in the receiving app. This copy is derived from the original descriptor.

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.

4.4.2 Object graph library

As mentioned in Section 3, 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 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.

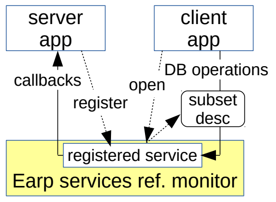
An app can use this library to conveniently construct a subgraph from an entry object that has capability or non-capability relationships with other objects. The lightweight nature of subset descriptors allows the library to proactively create descriptors as the app is performing queries. Internally, 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).

5 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 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 tables* and clients use



Earp translates DB operations into service callbacks

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

Figure 4: Inter-app services in Earp.

subset descriptors to access this "database." Defining virtual tables via callbacks is a standard idea, and a similar mechanism exists in SQLite [42]. 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 (§4). 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.

5.1 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.

- `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 (§7.1), 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.

5.2 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.

5.3 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`$\xleftrightarrow{n:1}$`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.

6 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 [9]) added to the browser runtime and libraries.

6.1 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 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.

6.2 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

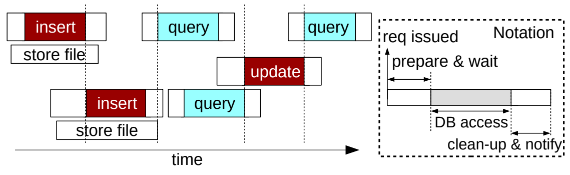


Figure 5: Constraints on request processing order in the thread pool.

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 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 5 shows an example of runtime request ordering.

6.3 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`.

6.4 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.

6.5 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 [17], 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.

7 Evaluation

7.1 Apps

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 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 [12], 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.

```
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 6: Policies defined for Elgg client apps, represented as JavaScript objects.

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`$\xleftrightarrow{1:n}$`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 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 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

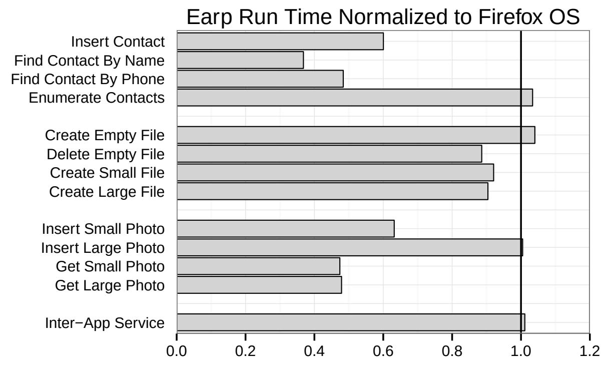


Figure 7: Microbenchmark results for storage and services. Smaller run time indicates better performance.

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.

7.2 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.

7.2.1 Microbenchmarks

We run various microbenchmarks to measure Earp's performance for storage and inter-app services. Figure 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 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.

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

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 7 shows that Earp performs roughly the same as the baseline, and the time spent for access control (result filtering) is negligible.

7.2.2 Macrobenchmarks

Table 1 reports end-to-end latency for several real-world workloads described in Section 7.1.

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 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 1, Earp achieves similar performance storing the emails in an app-defined database.

8 Related work

Fine-grained, flexible protection on mobile platforms. TaintDroid [13] is a fine-grained taint-tracking system for Android. Several systems [21, 40, 45] rely on TaintDroid for fine-grained data protection. Pebbles [40] is most related to Earp: it modifies Android's SQLite and XML libraries and uses TaintDroid to discover app-level structured data across different types of storage. Pebbles relies on developers using certain design patterns consistently to infer the structure of data and it is implemented in an app-level library, not in the platform. Pebbles can help cooperative apps avoid mistakes, like preserving an attachment of a deleted email, but, unlike Earp, it cannot confine uncooperative apps.

Many systems extend Android to support more flexible and expressive permission policies [3, 4, 10, 11, 15, 30, 31, 52, 53] or mandatory access control [6, 39]. FlaskDroid [6] provides fine-grained data protection by implementing a design pattern that lets content providers present different views of shared data to different apps. FlaskDroid is limited to SQLite-based content providers and does not support cross-table capabilities. By contrast, Earp's framework supports all types of services, including proxies for remote servers. Moreover, in contrast to all existing systems, Earp integrates access-control policies with the data model itself, via capability relationships.

Fine-grained protection in databases. Traditional access control systems for relational databases [5, 7, 16, 20, 26, 32, 35] are based on users or roles with relatively static policies. Recently, IFDB [37] showed how decentralized information flow control (DIFC) can be integrated with a relational database. IFDB also discusses foreign key issues, but focuses on potential information leakage due to referential integrity enforcement. This is a very different problem than the one solved by Earp's capability relationships. The key contribution of Earp is identifying the relational model as the unified foundation for protecting data storage and sharing on mobile platforms.

Protection on Web platforms. BSTORE [8] provides a file system API to Web apps and uses tags to enable flexible access control on files. It is similar to Earp in that access control is enforced by a central reference monitor regardless of where the resource is hosted (local or remote). Unlike Earp, BSTORE's data abstraction is unstructured files.

Several systems enable flexible policies [25, 29], controlled object sharing [28, 33], or confinement [22, 23, 43] for JavaScript in a Web browser. Earp puts protection much lower in the system stack. For Web code interacting directly with the OS and other apps, Earp provides a unifying abstraction for both storage and inter-app services and adds access control directly into the data model.

Native relational stores. Like Earp, there are previous efforts to make relational data directly supported by the OS, notably Microsoft's cancelled project WinFS [50]. WinFS contained a database engine to natively support SQL, and implemented files and directories 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 fits the current mobile development practice where databases are the de facto storage hubs [40]. Crucially, Earp uses an unmodified file system (unlike WinFS) to store blob data and to provide compatibility file APIs that have no performance overhead.

9 Conclusion

Earp is a new mobile app platform built on a unified relational model for data storage and inter-app services. Earp directly exposes fine-grained, inter-related structured data as platform-level objects and mediates apps' access to these objects, enabling it to enforce app-defined access-control policies with simple building blocks, both old (ACLs) and new (capability relationships). Earp securely and efficiently supports key storage and sharing tasks of essential apps such as email, contacts manager, photo gallery, social networking and cloud storage clients, etc.

Acknowledgments. We thank the anonymous reviewers, Simon Peter, Michael Walfish, and our shepherd, Brad Karp, for valuable feedback and suggestions. This work was partially supported by NSF grants CNS-1223396 and 140894 and the NIH grant R01 LM011028-01.

References

[1] Android developers: URI permissions. `http://developer.android.com/guide/topics/security/permissions.html#uri`. [Online; accessed 21-September-2015].

[2] Android Developers: Using content providers. `http://developer.android.com/training/articles/security-tips.html#ContentProviders`. [Online; accessed 21-September-2015].

[3] Backes, M., Gerling, S., Hammer, C., Maffei, M., and von Styp-Rekowsky, P. AppGuard – real-time policy enforcement for third-party applications. Tech. Rep. A/02/2012, MPI-SWS, 2012.

[4] Beresford, A. R., Rice, A., Skehin, N., and Sohan, R. MockDroid: Trading privacy for application functionality on smartphones. In *International Workshop on Mobile Computing Systems and Applications (HotMobile)* (2011), ACM.

[5] Bertino, E., Jajodia, S., and Samarati, P. Supporting multiple access control policies in database systems. In *IEEE Symposium on Security and Privacy* (1996).

[6] Bugiel, S., Heuser, S., and Sadeghi, A.-R. Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. In *USENIX Security Symposium* (2013).

[7] Byun, J.-W., and Li, N. Purpose based access control for privacy protection in relational database systems. *The VLDB Journal 17*, 4 (2008), 603–619.

[8] Chandra, R., Gupta, P., and Zeldovich, N. Separating web applications from user data storage with BSTORE. In *USENIX Conference on Web Application Development (WebApps)* (2010).

[9] CLOC – count lines of code. `http://cloc.sourceforge.net/`. [Online; accessed 17-September-2015].

[10] Conti, M., Nguyen, V. T. N., and Crispo, B. CRePE: Context-related policy enforcement for Android. In *Information Security Conference (ISC)* (2010).

[11] Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., and Wallach, D. S. Quire: Lightweight provenance for smart phone operating systems. In *USENIX Security Symposium* (2011).

[12] Elgg - open source social networking engine. `https://www.elgg.org`. [Online; accessed 17-September-2015].

[13] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2010).

[14] How I exposed your private photos - Facebook private photo hack. `http://www.7xter.com/2015/03/how-i-exposed-your-private-photos.html`. [Online; accessed 17-September-2015].

[15] Felt, A. P., Wang, H. J., Moshchuk, A., Hanna, S., and Chin, E. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium* (2011).

[16] Ferraiolo, D. F., Sandhu, R., Gavrila, S., Kuhn, D. R., and Chandramouli, R. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)4*, 3 (2001), 224–274.

[17] Georgiev, M., Jana, S., and Shmatikov, V. Rethinking security of Web-based system applications. In *International World Wide Web Conference (WWW)* (2015).

[18] Google Drive API for Android: authorizing Android apps. `https://developers.google.com/drive/android/auth`. [Online; accessed 18-September-2015].

[19] Google Drive API for iOS: authorizing requests of iOS apps. `https://developers.google.com/drive/ios/auth`. [Online; accessed 18-September-2015].

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

[21] Hornyack, P., Han, S., Jung, J., Schechter, S., and Wetherall, D. 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).

[22] Howell, J., Parno, B., and Douceur, J. R. Embassies: Radically refactoring the Web. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2013).

[23] Ingram, L., and Walfish, M. Treehouse: JavaScript sandboxes to help web developers help themselves. In *USENIX Annual Technical Conference* (2012).

[24] iOS developer library: App extension programming guide. `https://developer.apple.com/library/ios/documentation/General/Conceptual/ExtensibilityPG/index.html#//apple_ref/doc/uid/TP40014214`. [Online; accessed 17-September-2015].

[25] JAYARAMAN, K., DU, W., RAJAGOPALAN, B., AND CHAPIN, S. J. Escudo: A fine-grained protection model for web browsers. In *IEEE International Conference on Distributed Computing Systems (ICDCS)* (2010).

[26] JELOKA, S., ET AL. *Oracle Label Security Administrator's Guide*, release 2 (11.2) ed. Oracle Corporation, 2009.

[27] LI, T., ZHOU, X., XING, L., LEE, Y., NAVEED, M., WANG, X., AND HAN, X. 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).

[28] MEYEROVICH, L. A., FELT, A. P., AND MILLER, M. S. Object views: Fine-grained sharing in browsers. In *International World Wide Web Conference (WWW)* (2010).

[29] MEYEROVICH, L. A., AND LIVSHITS, B. Conscript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *IEEE Symposium on Security and Privacy* (2010).

[30] NAUMAN, M., KHAN, S., AND ZHANG, X. Apex: Extending Android permission model and enforcement with user-defined runtime constraints. In *ACM Symposium on Information, Computer and Communications Security (AsiaCCS)* (2010).

[31] ONGTANG, M., MCLAUGHLIN, S., ENCK, W., AND MCDANIEL, P. Semantically rich application-centric security in Android. *Security and Communication Networks 5*, 6 (2012), 658–673.

[32] OSBORN, S., SANDHU, R., AND MUNAWER, Q. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Transactions on Information and System Security (TISSEC)3*, 2 (2000), 85–106.

[33] PATIL, K., DONG, X., LI, X., LIANG, Z., AND JIANG, X. Towards fine-grained access control in JavaScript contexts. In *IEEE International Conference on Distributed Computing Systems (ICDCS)* (2011).

[34] RITCHIE, D. M., AND THOMPSON, K. The UNIX time-sharing system. *Communications of the ACM (CACM) 17*, 7 (1974).

[35] RIZVI, S., MENDELZON, A., SUDARSHAN, S., AND ROY, P. Extending query rewriting techniques for fine-grained access control. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2004).

[36] SALTZER, J. H. Protection and the control of information sharing in multics. *Communications of the ACM (CACM) 17*, 7 (1974).

[37] SCHULTZ, D., AND LISKOV, B. IFDB: Decentralized information flow control for databases. In *ACM European Conference in Computer Systems (EuroSys)* (2013).

[38] SHAHRIAR, H., AND HADDAD, H. Content provider leakage vulnerability detection in Android applications. In *SIN* (2014).

[39] SMALLEY, S., AND CRAIG, R. Security enhanced (SE) Android: Bringing flexible MAC to Android. In *Network and Distributed System Security Symposium (NDSS)* (2013).

[40] SPAHN, R., BELL, J., LEE, M. Z., BHAMIDIPATI, S., GEAMBASU, R., AND KAISER, G. Pebbles: Fine-grained data management abstractions for modern operating systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014).

[41] Sqlite foreign key support. `https://www.sqlite.org/foreignkeys.html`. [Online; accessed 18-September-2015].

[42] The virtual table mechanism of SQLite. `https://www.sqlite.org/vtab.html`. [Online; accessed 17-September-2015].

[43] STEFAN, D., YANG, E. Z., MARCHENKO, P., RUSSO, A., HERMAN, D., KARP, B., AND MAZIERES, D. Protecting users by confining JavaScript with COWL. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014).

[44] SUN, S.-T., AND BEZNOSOV, K. The devil is in the (implementation) details: An empirical analysis of OAuth SSO systems. In *ACM Conference on Computer and Communications Security (CCS)* (2012).

[45] TANG, Y., AMES, P., BHAMIDIPATI, S., BIJLANI, A., GEAMBASU, R., AND SARDA, N. CleanOS: Limiting mobile data exposure with idle eviction. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2012).

[46] HTML5 trumping iOS among app developers in emerging mobile markets. `http://www.zdnet.com/article/html5-trumping-ios-among-app-developers-in-emerging-mobile-markets/`. [Online; accessed 17-September-2015].

[47] Survey: Most developers now prefer HTML5 for cross-platform development. http://techcrunch.com/2013/02/26/survey-most-developers-now-prefer-html5-for-cross-platform-development/. [Online; accessed 17-September-2015].

[48] VIENNOT, N., GARCIA, E., AND NIEH, J. A measurement study of Google Play. In *ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (2014).

[49] WANG, R., XING, L., WANG, X., AND CHEN, S. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *ACM Conference on Computer and Communications Security (CCS)* (2013).

[50] Introducing "Longhorn" for developers, Chapter 4: Storage. https://msdn.microsoft.com/en-us/library/Aa479870.aspx. [Online; accessed 17-September-2015].

[51] XING, L., BAI, X., LI, T., WANG, X., CHEN, K., LIAO, X., HU, S., AND HAN, X. 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).

[52] XU, R., SAÏDI, H., AND ANDERSON, R. Aurasium: Practical policy enforcement for Android applications. In *USENIX Security Symposium* (2012).

[53] XU, Y., AND WITCHEL, E. Maxoid: Transparently confining mobile applications with custom views of state. In *ACM European Conference in Computer Systems (EuroSys)* (2015).

[54] ZHOU, Y., AND JIANG, X. Detecting passive content leaks and pollution in Android applications. In *Network and Distributed System Security Symposium (NDSS)* (2013).