Maxoid: Transparently Confining Mobile Applications with Custom Views of State

Yuanzhong Xu Emmett Witchel
The University of Texas at Austin

Abstract

We present Maxoid, a system that allows an Android app to process its sensitive data by securely invoking other, untrusted apps. Maxoid provides secrecy and integrity for both the invoking app and the invoked app. For each app, Maxoid presents custom views of private and public state (files and data in content providers) to transparently redirect unsafe data flows and minimize disruption. Maxoid supports unmodified apps with full security guarantees, and also introduces new APIs to improve usability. We show that Maxoid can improve security for popular Android apps with minimal performance overheads.

1. Introduction

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 provide security models to protect each app's private data, and control each app's access to shared data by specifying a variety of permissions. However, Android's 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 camera app to read a product bar code. In these examples, we call the app that needs to invoke a helper app the *initiator* and the invoked app the *delegate*, and we say that the delegate runs *on behalf of* the initiator. In Android, 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 §2.2). Instead of invoking an app, an initiator might use a third-party library, but the security issue remains if the library (built for general use) fails to protect the initiator's private data.

Mobile systems like Android offer new opportunities to secure mutually distrustful code. The app-based security model allows a new balance of usability and security for initiator and delegate apps that is not available to desktop or server systems. Android clearly distinguishes apps' private and public (shared) data. Based on that distinction and on Android's principled data abstractions (e.g., content providers), 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 applications and without requiring new, complex policies.

We propose Maxoid[1], a new security model that provides secrecy and integrity for mobile applications that invoke other applications. Maxoid allows delegates to access initiator private state, but prevents delegates from leaking these secrets to other applications 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 delegates. 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

EuroSys'15, April 21–24, 2015, Bordeaux, France.

http://dx.doi.org/10.1145/2741948.2741966

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

(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.

We summarize the major contributions of Maxoid below.

1. We introduce (§2) a model based on the private and public state of initiators and delegates, and analyze their security and usability requirements (§3).

2. Maxoid provides delegates with custom views of files (§4) and system content providers (§5) to achieve its security and usability goals. We describe our implementation of this approach in Android (§6).

3. We conduct case studies on popular Android apps (§2.2), using Maxoid to improve their security and measuring their performance (§7).

2. Background and Overview

Mobile platforms are usually app-centric. Because they run apps developed by mutually distrustful third parties, different apps are treated as separate principals carrying different access rights to resources on the device. A critical responsibility for the platform is to protect the private state of apps and to control their access to shared resources.

We describe the private and public state in Android, which is the most popular mobile platform, and is open source. Then we demonstrate the problem of processing sensitive data using untrusted apps via case studies, and how a naïve taint-tracking solution suffers from poor usability. Finally, we give an overview of Maxoid.

2.1 Private and public state in Android

In Android, each app is assigned a dedicated Unix 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.).

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

In earlier versions of Android, an app can either have no access to 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 on external storage 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.

2.2 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, and 2) apps that need help from data processing apps. One theme that emerges is that Android's access control model, while impressively fine-grained (e.g., per-URI permissions), provides no information flow control on sensitive data, which limits how effectively it can enforce security protections.

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 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 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

Category	# of Apps	Representative App	Operation	State left after the operation	
				Private state	Public state
Document viewer, editor, converter	17	Adobe Reader	open a file	XML: recent files.	A copy of the file on SD card when opening a content URI.
		Kingsoft Office	open a file	ADF : recent files.	A copy of the file on SD card when opening a content URI. A thumbnail for this file on SD card. Entries in a database stored in SD card.
Scanner	20	Barcode Scanner	scan a QR code	DB: recent scans.	
		CamScanner	scan a file	DB: recent scans.	An image file saved to SD card. A thumbnail for this file on SD card. A log file on the SD card.
Photo	30	CameraMX	take a photo		The photo file saved to SD card. A new entry in Media Provider.
			edit a photo		A new entry in Media Provider.
Media	10	VPlayer	play a video	DB: playback history.	A thumbnail for this video on SD card.

Table 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 a SQLite database; ADF indicates state saved in files with app-defined formats.

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 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.

2.3 Taint tracking and challenges

Additional information flow control mechanisms are needed to secure the use cases in §2.2. 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 [8].

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 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.

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 [10] is a fine-grained taint tracking system with moderate overhead on Android, but it does not track implicit data leakage via control flows.

2.4 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 §3.1).

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 §3.4).

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 [18].

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 – an IPC mechanism for an app component to invoke another.

2.5 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. 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.

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 the principle of *minimum isolation*: whenever a data flow is safe, it should be allowed. In addition to minimum isolation, Maxoid strives to be backward compatible. Minimum isolation guides ***U1***and ***U2***, while backward compatability 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., C^A).

[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.

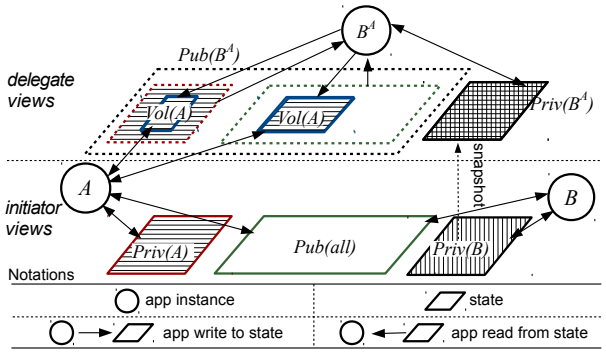

Figure 1: Overview of Maxoid confinement. Hatching in a state box indicates taints: $Priv(A)$ and $Priv(B)$ are the sources of taints, $Vol(A)$ is tainted by $Priv(A)$, and $Priv(B^A)$ is tainted by both $Priv(A)$ and $Priv(B)$.

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 $Pub(all)$.

3.1 Confining delegates by custom views

Figure 1 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 §3.3).

Information flows. A directed path of solid arrows in Figure 1 represents an information flow. Maxoid doesn't use fine-grained taint tracking [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 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 2 demonstrates how $pPriv$ and $nPriv$ evolve over time.

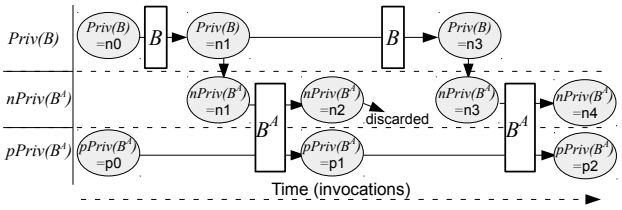

Figure 2: Normal and persistent private states evolving over time. A solid box is an app instance running for a period. An ellipse shows the value (version number) of a state before or after an invocation.

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

3.3 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.

3.4 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.

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 IPC is usually 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***).

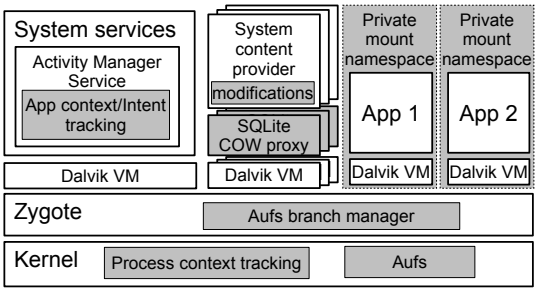

Figure 3: Maxoid system architecture. Gray boxes are new components or modifications to Android.

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 §6.1.

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 (§6.3).

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.

3.5 Maxoid system architecture

The system architecture of Maxoid is shown in Figure 3. 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 (§3.4, §6.2). Other components implement Maxoid view switching for file system (§4) and system content providers (§5). Zygote is the parent process in Android that forks all app processes, which preloads common Java classes and resources, to speed up application launching.

4. File System

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

4.1 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 4 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.

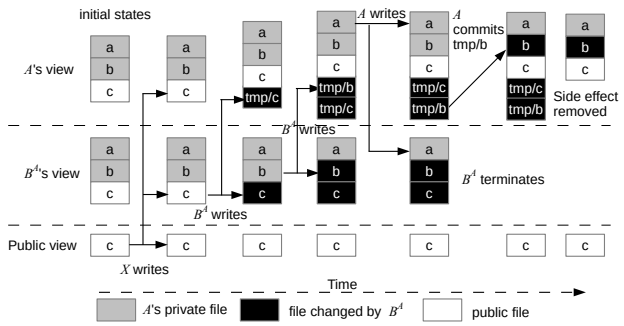

Figure 4: Views of files for A, B^A and X. The figure shows a scenario where A wants B^A to edit a file *b*, but B^A also has side changes on file *c*.

B^A can 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.

4.2 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 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) 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.

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

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 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.

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 §3.2, 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 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 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.

5. System Content Providers

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

5.1 System content providers in Maxoid

System content providers, like Downloads, Media, Contacts and Calender, are another type of public resource, and potentially sources of serious data leaks [43]. They map URIs to data, and support 4 operations – insert, update, query and delete – on each URI. They typically use SQLite databases as backends. We built a copy-on-write proxy layer (§5.2) 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 (§5.2). 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.

5.2 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 5 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 (§3.3), 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.

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 6). 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

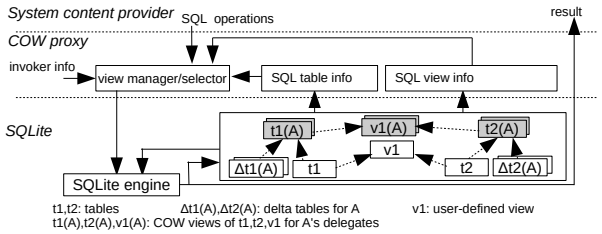

Figure 5: COW proxy interacts with the content provider and SQLite. Note that `v1` is a SQL view defined by the content provider.

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

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 6. Its definition satisfies the constraints for SQLite's subquery flattening optimization [34], which makes queries on it efficient because the query planner moves the `WHERE` clause (if any) on this view into the two inner subqueries.[5]

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.

[5] SQLite version 3.7.11 (as used in Android 4.3.2) does not do subquery flattening on the `UNION ALL` view if the query has an `ORDER BY` clause, unless the query uses "*" as the columns. Version 3.8.6 partially fixed this issue, but still requires the `ORDER BY` columns to be a subset of the columns being queried. SQLite maintainers confirmed this issue but have not started to fix it. Therefore, we ported SQLite 3.8.6 to Android 4.3.2, and our proxy adds `ORDER BY` columns to query columns when necessary.

To support update and delete, we define `INSTEAD OF` triggers on the per-initiator COW views (Figure 6). 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 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 5), 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.

5.3 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.

6. API and Implementation

We implement Maxoid by modifying Android 4.3.2.

6.1 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 (§4.2) 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 (§6.3).

3. An initiator can manage its volatile state (§4 and §5).

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 (§7.1).

APIs for delegates. First, Maxoid introduces persistent private state, which is a directory in internal storage (`/data/-data/ppriv/⟨package_name⟩`) (§3.2, §4.2). 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.

6.2 Tracking app execution context

§4 and §5 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 §3.4). 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 §4).

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 [13]). 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 §5). 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.

6.3 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.

7. Evaluation

We first show how to use Maxoid to improve security for apps discussed in §2, then measure performance of Maxoid.

7.1 Maxoid use cases

Out of the 77 data processing apps we analyzed in §2, 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 $Vol(\text{Dropbox})$ 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 (§6.3), 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 §3.2, 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 $pPriv$.

7.2 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 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.

7.2.1 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. 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

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

Setup	CPU-bound operations	Internal File System						User Dictionary Provider				
		4KB files			1MB files			insert	update	query 1 word	query 1k words	delete
		read	write	append	read	write	append					
initiator	0	0						1.3%	0.4%	0.5%	0.2%	1.0%
delegate	0	7.5%	31.7%	58.7%	4.8%	18.1%	52.8%	8.1%	16.1%	5.6%	13.7%	17.3%

Table 3: Microbenchmark overheads compared to Android. Results are averaged over 1000 trials. **CPU-bound operations**: No overhead. **Internal file system**: No overhead for initiators. Read – read files; Write – create and write to files; Append – append to the original files to double their sizes. **User Dictionary Provider**: 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.

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 4: Times for 1) downloading 100 1KB files, and 2) scanning 100 780KB image files and storing the metadata to Media Provider. Results are averaged over 5 trials.

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. 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.

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 4 shows the result, where 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.

7.2.2 Application benchmarks

We measure the latency of performing several application-specific tasks, as listed in Table 5. 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.

8. Discussion

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

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 5: User-perceivable latency of performing various tasks using different apps. Results are averaged over 5 trials.

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.

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 (§3.4). 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.

9. Related Work

Invoking untrusted code on sensitive data is a classic security problem that has been addressed by several desktop/server systems. Many approaches exist, but one family uses information flow such as language-level decentralized information flow control (DIFC) [1, 5, 19, 23, 35], OS-level DIFC [9, 15, 17, 42], PL-OS DIFC [28] and architectural-OS information flow [38]. Another approach uses access control such as mandatory access control (MAC) [21, 29] or capabilities [40]. Compared to these approaches, Maxoid solves usability issues for legacy applications while providing strong security guarantees.

Taint tracking. TaintDroid [10] is a taint tracking system for Android, which detects data leakage in a much more fine-grained way than Maxoid. There are also systems that leverage TaintDroid [13, 36] to prevent data leakage, but they do not aim at providing safe invocation of untrusted delegates. TaintDroid cannot detect implicit data leaks through control flows, and does not support fine-grained taint tracking for native code. In contrast, Maxoid is more conservative, i.e., a delegate's (even in native code) output is always controlled.

Flexible permission granting and MAC. Recent systems [2, 3, 6, 24, 25, 41] have been proposed to support flexible runtime permission granting or revoking, in addition to Android's install-time permission assignment. SE Android [32] and FlaskDroid [4] provide mandatory access control. Systems like ServiceOS [22], Bubbles [37], IPC Inspection [11] and QUIRE [7] provide applications different privileges when they execute in different contexts. While these systems improve security by enforcing stronger control, they cannot ensure confidentiality when the user wants to use untrusted apps to process sensitive data.

Using libraries instead of apps. Instead of invoking delegate apps, an app can incorporate third-party libraries to extend its functionality. Unfortunately, these third-party libraries are still untrusted [20]. A library is less secure than a separate app because, as part of the same app, it has access to the app's entire private state. Moreover, adding functionality to a single app can add to permission requirements for that app, which is undesirable for security.

Networked systems. Sandboxing and information flow control for the server side of untrusted apps have been proposed for a mobile platform [18], and social networking platforms [12, 31, 39]. Similar approaches might help address the limitation that Maxoid must block network for delegates. However, they require a trusted cloud to host all third-party apps, which does not yet exist for Android or iOS.

AdSplit [30] and AdDroid [26] split an app and its advertising into separate processes for security. Similar approaches can enable Maxoid delegates to use advertising.

Using different views of data for security. Solitude [14], Apiary [27] and Mbox [16] 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 different from them and suited for mobile apps' collaboration on sensitive data. Pebbles [33] and its application PebbleDIFC can present different views of data to apps, e.g., the user can prevent an untrusted app from accessing a sensitive photo. They require the user to mark sensitive data, instead of having the state model for initiators and delegates in Maxoid.

10. Conclusion

We have presented Maxoid, a system to improve the security of Android applications that collaboratively operate on sensitive data. Maxoid achieves security and usability by maintaining a custom view of state for each app. It differs from previous sandboxing mechanisms by tolerating a wider range of app behavior while maintaining security.

11. Acknowledgements

We thank the anonymous reviewers, Alan M. Dunn, Michael Z. Lee, and our shepherd, Andreas Haeberlen, for valuable feedback and suggestions. This research is supported by CCF-1333594, CNS-1228843, and R01 LM011028-01.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] Petros Efstathopoulos and Eddie Kohler. Manageable fine-grained information flow. In *ACM European Conference in Computer Systems (EuroSys)*, 2008.

[9] 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.

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

[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.

[12] Daniel B Giffin, Amit Levy, Deian Stefan, David Terei, David Mazieres, John Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[13] 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.

[14] 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.

[15] 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*.

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

[17] 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.

[18] 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.

[19] 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.

[20] Benjamin Livshits and Jaeyeon Jung. Automatic mediation of privacy-sensitive resource access in smartphone applications. In *USENIX Security Symposium*, 2013.

[21] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the linux operating system. In *USENIX Annual Technical Conference (ATC)*, 2001.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

[28] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical fine-grained decentralized information flow control. In *ACM SIG-*

PLAN Conference on Programming Language Design and Implementation (PLDI), June 2009.

[29] Indrajit Roy, Srinath Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. Airavat: Security and privacy for MapReduce. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2010.

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

[31] Kapil Singh, Sumeer Bhola, and Wenke Lee. xBook: Redesigning privacy control in social networking platforms. In *USENIX Security Symposium*, 2009.

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

[33] 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.

[34] The SQLite Query Planner. `http://www.sqlite.org/optoverview.html`.

[35] Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazières. Protecting Users by Confining JavaScript with COWL. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[36] 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.

[37] 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.

[38] 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.

[39] Bimal Viswanath, Emre Kiciman, and Stefan Saroiu. Keeping information safe from social networking apps. In *Proceedings of the 2012 ACM workshop on Workshop on online social networks*, 2012.

[40] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for unix. In *USENIX Security Symposium*, 2010.

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

[42] 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.

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