Masquerade of Mobile Applications: Introducing Unlinkability in a Practical Way

Huan Feng and Kang G. Shin
Department of Electrical Engineering and Computer Science
The University of Michigan – Ann Arbor
{huanfeng, kgshin}@umich.edu

***Abstract*—Smartphone apps are becoming a popular vehicle to collect users' personal interests, demographics and other private information. Due to lack of regulation, a curious party can covertly link and aggregate sensitive information from independent sources (sessions or apps) over time to conduct unwanted user profiling, targeted advertising or surveillance. Such *unregulated aggregation* is rooted at the non-existence of unlinkability in the mobile ecosystem. On one hand, the mobile ecosystem is over-populated with various persistent identifiers and fueled by the abundance of user information; on the other hand, users only expect app usages that are functionally-dependent to be linkable. To bridge this gap, we propose a practical solution, called Mask, that allows users to negotiate to what extent his behavior can be linked and aggregated. Specifically, Mask introduces a set of *private execution modes* which enable different levels of unlinkability. Mask is a user-level solution and does not require any change in the existing ecosystem, thus allowing for easy deployment. We present the technical details and challenges of our user-level implementation and evaluate its runtime performance as well as applicability.**

I. Introduction

During the past decade, we have been moving swiftly towards a mobile-centered world and smartphones have become the very hub of mobile user information and activities. Compared to traditional desktop/laptops, it is continuously connected to the Internet, always carried by the owner and rarely shared among different users. Moreover, a smartphone is equipped with various powerful sensors, becoming able to peek into the physical environment surrounding the users. All of these make smartphones an ideal vehicle for user tracking and profiling.

There have already been numerous studies on how to stop a malicious party (an app or third party within the app) from accessing the information it shouldn't access [1–5]. Orthogonal to these studies, we address an emerging privacy threat imposed by a curious party who covertly links and aggregates a user's behavioral information collected from independent sources — across sessions and apps — without his consent or knowledge. In the current smartphone ecosystem, curious parties can be:

- *Mobile apps*. For example, a user follows political news and religious articles using the same news app like CNN or NYTimes. By aggregating both the user's political and religious interests, the app can deliver him personalized news content, such as "The END Of Anti-Gay Religious Rhetoric in Politics." However, if the user is sensitive about what he reads, he may not want this type of unsolicited correlation across his interests in different subjects.
- *Advertising agencies*. For example, a user downloads two ad-powered apps and exposes his sexuality to the first one and his location to the second. However, since these apps include the same ad library, the advertising agency can associate both the sexuality and the location with him and send him targeted dating advertisements. The user, on the other hand, does not approve this type of covert aggregation.
- *Network sniffers*. As recently publicized in the news media, government agencies such as US NSA and GCHQ often conduct public surveillance by sniffing network traffic and aggregating personal information leaked by smartphone apps and ad libraries. A recent study [6] shows that a similar sniffer is able to attribute up to 50% of the mobile traffic to the 'sniffed' users, on top of which detailed personal interests, such as political views and sexual orientations, can be extracted.

The severity and prevalence of this threat are rooted at the nonexistence of unlinkability in the smartphone ecosystem. By exploiting various levels of consistency provided by device identifiers, software cookies, IPs, local and external storages, an adversary can easily correlate app usages of the same user and aggregate supposed-to-be 'isolated islands of information' into a comprehensive user profile, irrespective of the user's choice and (dis)approval.

However, from the user's perspective, only app usages that are functionally-dependent should be linkable. For example, for GTalk, app usages under the same login should be linkable to provide a consistent messaging service. For Angry Birds, usage of the same app should be linkable to allow the user to resume from where he stopped. In contrast, for most query-like apps, such as Bing and Wikipedia, which neither enforce an explicit login nor require consistent long-term 'memories', app usages should be globally unlinkable by default.

To bridge this gap, we propose Mask, the first user-level solution that allows the user to negotiate to what extent his behavior can be linked and aggregated. A user-level solution is essential because the entire mobile ecosystem is fueled by the abundance of user information and any solution that requires cooperation/modification of the OS/ecosystem is unrealistic. Specifically, Mask introduces a set of *private execution modes* that allow the users to maintain multiple isolated profiles for each app. Each app profile can be temporary, being recycled after each session, or enduring, persists across multiple sessions. Upon invocation of each app, a user can apply one of the following modes to the current *app session* (from the start to termination of the app), according to his usage scenario. If he wants to:

- Use this app while logged in, choose the identifiable mode. All app usages under the same login are now linkable and the app delivers uncompromised personal services while disallowing aggregation across unrelated apps.
- Keep states or use the states saved before, apply the pseudonymous mode. In this mode, a user can maintain multiple profiles and only app usages in the same profile are linkable.
- Execute this app without leaving any trace, apply the anonymous mode. Each session is treated as independent and app usages are confined within the current session.

All user behaviors originate from the mobile apps, either directly or indirectly. By enabling the aforementioned private execution modes which isolate app usages at this very source, Mask provides a client-side solution without requiring any change to the existing ecosystem. We have implemented Mask on Android at user level, without requiring any modifications on the Android framework. This is achieved by enforcing a lightweight user-level sandbox that creates an isolated runtime environment with stripped account information, anonymized device IDs & software cookies, and isolated persistent storage. Our solution is able to bring privacy benefits to more than 70% of the apps and incurs negligible overhead in the app's runtime performance. Our user study on 27 users find that more than 60% of them find it useful to maintain multiple isolated profiles for mobile apps and 11 of them are willing to use this feature on a daily basis.

The rest of this paper is organized as follows. The next section introduces the background of unregulated aggregation of mobile app usages and lists the practical challenges encountered. Section 3 presents a high-level design of Mask, while Section 4 describes our user-level implementation on Android as well as some evaluation results. Section 5 covers the related work, and finally Section 6 concludes the paper.

II. Background & Challenges

A. Unregulated Aggregation

In current mobile ecosystems, an interested/curious party can covertly link and aggregate app usages of the same user over time, without his consent or knowledge, which we call *unregulated aggregation* of app usages. Here, we describe three major adversaries — mobile apps, A&A agencies, and network sniffers — and show how they aggregate app usages in practice.

1) Smartphone Applications: Smartphone apps aggregate users' app usages mainly for personalization. By tracking app usages over time and feeding them to domain-specific mining/learning algorithms, smartphone apps can deliver contents tailored to each user. Even if a user doesn't give an explicit consent (by logging in), apps can still identify and aggregate usages of the same user. In fact, if only for the purpose of user tracking, mobile apps have options far easier and simpler than enforcing login. Specifically, a smartphone app can use device IDs or system IDs, such as IMEI and Android ID, as a consistent user identity to aggregate his app usages remotely on the server, or exploit the consistent & persistent storage on the device and achieve the same goal locally.

2) Advertising & Analytics Agencies: To enable targeted advertising, A&A agencies are also interested in aggregating personal interests and demographics disclosed in app usages. Specifically, app developers include clients of these ad agencies—ad libraries—into their apps and proactively feed sensitive information requested by these libraries [7]. Moreover, since an ad library shares the same permission with its host app, it can also access and collect private information on its own. To identify and aggregate information of the same user, third-party libraries embed user identifiers into the traffic they send to the back-end servers. Such a user identifier can be the hash value of a device/system ID or a local cookie. These A&A agencies can be more dangerous than smartphone apps as they can aggregate usage behaviors across multiple apps carrying the same library.

3) Network Sniffers: Unlike the aforementioned parties, a network sniffer cannot collect information directly from the user's device on its own and can only extract information from raw network traffic. Moreover, from the sniffer's perspective, the network traffic can be really messy: some are directly marked with the actual device ID, some are tagged with hashed ones, some only embed app-specific ID (such as a craigslist user ID) while some others are encrypted and completely useless. However, as MOSAIC [6] shows, by exploiting the relative consistency of IP, it can associate different IDs that represent the same user. This way, even the traffic marked with different and seemingly unrelated user IDs can be aggregated. As publicized recently, a similar technique is used by government agencies (e.g., US NSA and GCHQ) for public surveillance.

In summary, these parties have an increasing scope of information collection and aggregation, and decreasing control on the client side (mobile device). Besides, they're not independent parties, but operate more like subordinates of an integrated adversary. Fig. 1 illustrates the information flow among these parties.

B. Practical Challenges

The following practical challenges need to be addressed when developing a solution.

1) Intermingled Interests of Different Parties: Free apps dominate mobile app stores with 91% of the overall downloads [8], and app developers include ad libraries to monetize these free apps. Therefore, smartphone apps share the same financial interest with A&A agencies, and should thus not be trusted. In fact, smartphone apps may deliberately collude with A&A agencies and feed them user demographics, such as age

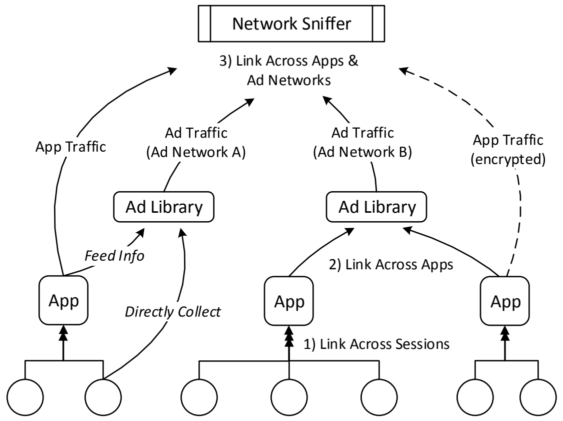


Fig. 1. How usages in different *app sessions* (circles in the figure) are collected, aggregated and propagated by smartphone apps, A&A agencies and network sniffers.

and gender, which ad libraries wouldn't be able to know on their own. On the other hand, OS vendors, whose popularity highly depends on the activeness of app developers, are reluctant to add privacy-enhancing features that may undermine the app developers' financial interests. Therefore, any defense that requires extensive OS-level modifications/cooperations will be impractical and not deployable.

2) Overpopulated User Identifiers: Unlike the web case where users are usually identified and tracked using cookies, smartphone apps have much more choices. Exploiting the consistency provided by numerous device and system IDs (some of which do not even require a permission to access), they can track app users both consistently and persistently. Moreover, since apps have arbitrary control over their persistent storage, they can perform local aggregation of users' information which doesn't even require any type of ID.

3) Trade-off between Functionality & Privacy: Enhancement of privacy often implies sacrifice of functionality. Similarly, opt-out unregulated aggregation, while providing a better privacy guarantee, also impairs personalized user experience. This trade-off, instead of being context-invariant, is subject to an app's nature as well as the user's preference. Thus, one must make a useful and adjustable trade-off between privacy and functionality.

These fundamental issues greatly reduce the set of tools and techniques a practical solution can use, rendering most existing proposals ineffective in practical settings (see Section 5 for the state-of-art).

III. MASK: A CLIENT-SIDE DESIGN

A. Basic Design Idea

The basic idea behind Mask's design is to allow only those app usages that are functionally dependent on each other to be linkable while keeping others unlinkable by default. This is achieved by introducing a set of *private execution modes* through which app users can provide explicit consent, on whether and within which scope the app usages in current *session* can be aggregated. The private execution modes are introduced based on our observation of how apps are actually used. Specifically, we first classify app-usage scenarios according to the levels of linkability required by app functionality and then introduce different private execution modes according to these app usage patterns.

B. App Usage Patterns

In Mask, the basic unit of a user's app usage is *session*, which represents a series of continuous active interactions between the user and an app to achieve a specific function. On Android, this typically corresponds to the activities between the invocations of function calls `onCreate` and `onDestroy`. The duration of a session is relatively short. Therefore, personal information in a single session can be very limited, and hence, different parties are devoted to linking and aggregating different sessions of the same user.

Mask classifies app usage patterns depending on whether and to what extent app usages in different sessions should be linkable. The three app usage patterns introduced below—*stateless*, *durative* and *exclusive*—characterize app's increasing need on the level of consistency in its runtime environment.

1) Stateless Pattern: the user's activity in one session does not depend strongly on the states of, and information from other sessions. By 'not strongly', we mean the the app is able to deliver its main functionality without any information from previous sessions, possibly at the expense of reducing optional personalized features. A wide spectrum of apps fit this pattern, including query-like apps such as Wikipedia and Yelp, apps from most news media such as NYTimes and CNN, and simple games such as Doodle Jumps and Flappy Bird.

2) Durative Pattern: the user requires persistent states and long-term 'memories' of an app to perform his current activities, but does not need to reveal his real-world identity. This pattern fits note-keeping apps, music player, books & magazines, complex games with a storyline or levels that need to be unlocked (such as Angry Birds), and etc.

3) Exclusive Pattern: the user must execute the app with an explicit identity, such as a user ID or account, and is willing to take the accompanied privacy risk. It covers most of social apps, such as Facebook and Twitter, as well as communication apps including WhatsApp, Yahoo Messenger, etc. If an app fits this pattern, the corresponding usages can and should be linkable across all the sessions that share the same account.

Note that the different users might apply different patterns when using the same app. This is subject to the preference of each specific user.

C. Aligning Usage Pattern with Privacy

Let's first discuss the default scenario in a contemporary mobile OS, using Android as an example. An app can track a user via device IDs such as IMEI or MAC address, which typically require permission, or via system IDs such as SERIAL number and android ID, which do not require any permission at all. If an app or an ad library wants, it can always export a cookie to its local storage, or more persistently, to external storage. These persistent anchors in the app's runtime allow

TABLE I
THE GAP OF LINKABILITY: EXPECTATION VS. REALITY

Scenario	*Linkable*	
	Across Sessions	*Across Apps*
stateless	single	single
durative	some - all	single
exclusive	all	single - some
default (reality)	all	all

an adversary to link and aggregate usage across all apps and sessions. However, from the user's perspective, this linkability is far too strong for most app functionalities.

For apps executed *statelessly*, linkability is not needed even in the weakest form since each session is inherently independent. For apps executed *duratively*, linkability is only needed across (some, not necessarily all) sessions of the same app—for example, a user may wish to maintain two separate instances for the same gaming app. Even for apps executed *exclusively* there are additional privacy issues. When a user executes an app with login, its app usages should only be linkable within the app, or at most across apps using the same login — instead of across all sessions and apps. Table I summarizes this gap between expectation and reality.

D. Private Execution Modes

Having understood the user's requirement on linkability, we present an intuitive way for the user to give explicit consent. Specifically, Mask introduces a set of *private execution modes*. Whenever a user starts an app, he can choose which mode to apply on the current session, implicitly specifying whether and within which scope his app usages can be aggregated.

Mask provides three types of private execution modes—identifiable, pseudonymous and anonymous—which are mapped to the three usage patterns we defined earlier and provide increasing levels of unlinkability of the user's app usages. An ordinary user can make this decision by following a few simple rules, without any domain-specific knowledge. Specifically, when an app starts execution and the user wants to:

- Use this app while logged in, he should choose the *identifiable mode*. Assuming all app usages under the same login are linkable, Mask allows an app to deliver uncompromised personal services while disallowing aggregation across unrelated apps.
- Keep states or use the states saved before, he should apply the *pseudonymous mode*. In this mode, a user can maintain multiple context-based profiles and only app usages in the same profile are linkable.
- Execute this app without leaving any trace, he should apply the *anonymous mode*. Each session is treated as independent and app usages are confined within the current session.

Fig. 2 presents an overview of Mask's design. Note that our design is not restricted to any specific mobile OS or platform since its rationality is rooted in the general notions of app usage patterns. Different choices of Mask's implementation only reflect emphasis on different aspects, such as performance, robustness or practicality.

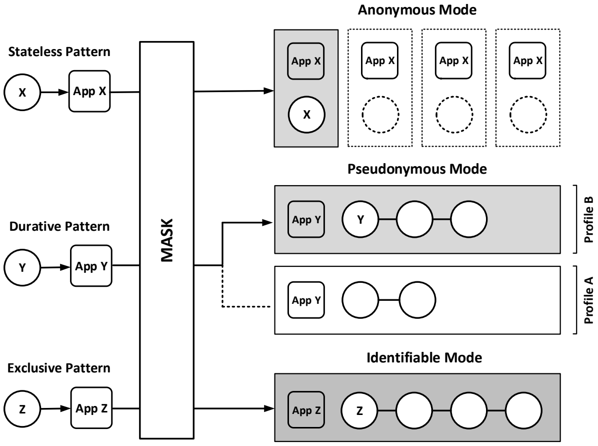


Fig. 2. Mask provides different *private execution modes* for each usage pattern we classified and only app *sessions* that are functionally dependent on each other are linkable.

IV. IMPLEMENTATION & EVALUATION

The principle that drives every decision in our implementation is deployability without any unrealistic dependencies or assumptions on other parties, such as platform-level support or collaboration with A&A agencies. This is important because (1) the privacy threat under consideration is prevalent and needs to be dealt with urgently and users should be given a choice to opt out right away; and (2) as we discussed earlier, there exist intermingled benefits among different parties in the current mobile app ecosystem and counting on any of them may degrade the practicality of a solution. Guided by this principle, we developed a client-side prototype of Mask on Android (4.1.1) at the user level. Next, we provide the technical details and challenges of our user-level implementation.

A. User-Level Sandbox

To enable the aforementioned private execution modes, we need to provide an isolated runtime environment. Since practicality is priority in our implementation and users are less likely to use a custom ROM or root their device solely for privacy protection, we need a user-level sandbox implementation.

As proposed by the system communities [9], the dynamic linking process can be exploited to support program customization. Any dynamically linked executable keeps a mapping between external function symbols and the corresponding memory addresses, known as the *global offset table* (GOT). By rewriting entries in the GOT, access to any external function symbols can be redirected to a user-specified function. This makes it possible to intercept library calls in user-level and deliver security and privacy features [10].

We adopt this user-level technique to achieve two goals: intercepting inter-process communications (IPCs) between system services and apps to strip personal and device identifying

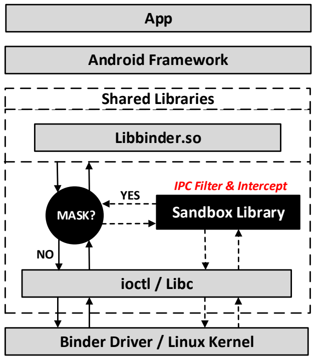


Fig. 3. How to achieve Binder IPC interception.

information that enable aggregation at the server side; and provide an isolated per-sandbox storage to break local (on-device) aggregation.

1) IPC (Intent) Interception: IPC is the only supported mechanism in Android that allows an app to interact with other processes and exchange information. Any explicit communication, using *Intents*, or implicit ones, such as getting information from system services using high-level APIs, are supported by this IPC mechanism. To strip personal and device identifying information an app could get, we need to be able to intercept, understand and modify any IPCs between this app and other parties. This brings some technical challenges and requires a good understanding of how IPC works in Android.

In Android, the design of IPC, `Binder`, is conceptually a lightweight RPC mechanism which allows one process to invoke routines in another process. It consists of two components: the shared library `libbinder.so` in user space and the Binder driver in kernel space. They communicate with each other according to the Binder protocol via the bionic libc call *ioctl*. All high-level objects such as *Intents* are packed into a container object (Parcel) and then sent through the binder protocol as a byte array. By intercepting the *ioctl* function call in `libbinder.so`, we can exercise arbitrary user-level control. Specifically, we overwrite the GOT in `libbinder.so` and redirect *ioctl* to a wrapper function. This wrapper allows us to intercept both incoming and outgoing communications, both are indispensable to achieve our goal. Intercepting outgoing traffic lets us know what request this app sends while intercepting the incoming traffic allows us to change the results returned. Fig. 3 summarizes this process.

On top of this interception mechanism, we can impose control over any intra- or inter-app communications as well as the app's interactions with system components. Here, we focus on the latter because in Android, identifying information is centrally managed by system services. Table II summarizes the list of identifiers Mask anonymizes. It contains the most commonly-used IDs but may not be a complete list of all potential identifying information. However, the associated technical underpinning is general enough to cover other identifiers, if necessary. We also exclude quasi-identifiers, such as IP or location, because compared to the the explicit identifiers addressed in Mask, they are far less consistent and reliable [6, 11], especially in the mobile context. Moreover, there are already independent lines of research on these subjects, such as IP anonymization and location anonymity.

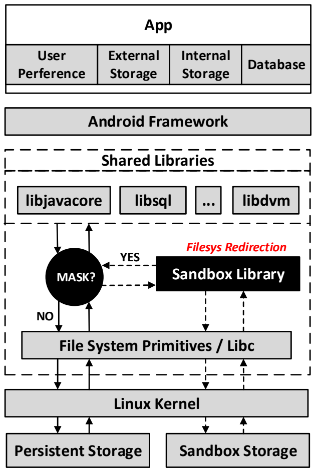


Fig. 4. How to achieve persistent storage redirection/isolation.

TABLE II
LIST OF COMMONLY-USED IDENTIFIERS MASK ANONYMIZES

ID	*System Service*	*Permission*
IMEI/MEID	iphonesubinfo	READ_PHONE_STATE
SUBSCRIBER ID	iphonesubinfo	READ_PHONE_STATE
PHONE NUMBER	iphonesubinfo	READ_PHONE_STATE
MAC ADDR	wifi	ACCESS_WIFI_STATE
ACCOUNTS	accounts	GET_ACCOUNTS
ANDROID ID	settings	NONE
SERIAL	settings*	NONE

2) Persistent Storage Isolation: Isolating persistent storage for each sandbox is necessary because it prevents local aggregation of the user's app usages, and also break the consistency of software cookies. Android provides the following options for persistent storage: Shared Preferences, Internal Storage, External Storage and SQLite Databases.

All of these storage options are built upon file system primitives provided by `Bionic Libc`, such as *open*, *stat*, *mkdir*, *chmod*, etc. By intercepting these primitives and modifying the corresponding input parameters, we can exercise arbitrary control over the app's interactions with the file system. Specifically, we create shadow directories which resemble the initial states of the app for each sandbox upon its creation, and then redirect all upcoming file system operations from the app-specific directories to the corresponding shadow directories. Figure. 4 illustrates this process.

B. Sandbox Manager

So far, we have introduced how a user-level sandbox is implemented to provide an isolated runtime environment. Next, we describe how these sandboxes are created, executed and destroyed to deliver the *private execution modes* in Mask.

1) Sandbox Management: The lifecycle of each sandbox is centrally controlled by a sandbox manager. The sandbox manager maintains a meta file for each sandbox, which contains sandbox-specific parameters: paths to the designated shadow directories, anonymized values of the persistent identifiers. When a sandbox is created, the manager generates a sandbox-specific meta file and allocates it shadow directories both in the local storage and the SD card. Then, any resources required for the initial states of this app will be copied into the shadow directories. The sandbox is then ready to start. When a sandbox is executed, the manager will initialize the runtime with information stored in the sandbox's meta file and activate the library-call interception mechanism. When a sandbox is destroyed, the sandbox manager first clears the local storage designated for the sandbox and then deletes the corresponding meta file.

Whenever an app is executed in anonymous mode, a new sandbox is created and applied; the sandbox will be immediately destroyed after the current session terminates. Note that, in Android, the termination of a session (`onDestroy`) is automatically controlled/optimized by the system. The user can force the current session to terminate by removing the app from the recent apps list. If an app is executed in pseudonymous mode, the user can choose to reuse an existing sandbox or create a new sandbox; a sandbox will only be destroyed when a user explicitly specifies it. If an app is executed in identifiable mode, a designated sandbox gets initialized; the user always runs the same sandbox.

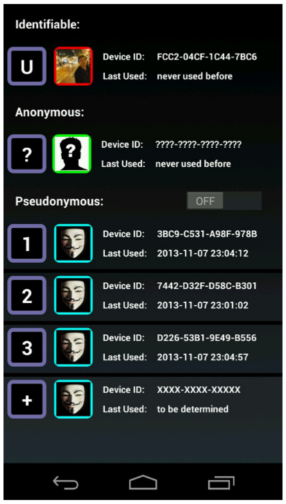
2) Multi-process Support: For an app with only one process, its execution in a sandbox is simple; but for apps with multiple processes, it can be complicated. Since what we implemented is a per-process sandbox and Android allows an app to host multiple processes, an app will crash if different processes are executed with inconsistent runtime environment. Therefore, we equip our sandbox with multi-process support. Each time an app process starts, the sandbox manager will first tell whether a sandbox is already created for this app (by maintaining a lock file in this app's local storage). If so, the new process will join the existing sandbox and share the same runtime environment.

3) UI Design: The logic of sandbox manager is hidden behind an intuitive UI. Mask's UI is displayed right before the launcher activity of an app starts, and is executed as an independent process isolated from all other components of the app. It offers a nice and intuitive way for end-users to manage profiles without revealing too much details. As shown in Fig. 5, Mask provides three *execution modes*—identifiable, pseudonymous and anonymous. For identifiable and anonymous modes, the user can simply click-and-use; for pseudonymous mode, Mask allows the user to maintain multiple profiles on the same device.

C. APK Rewriter

So far, we described how to build a user-level sandbox in Android that provides unlinkable runtime environments. Next, we describe how to merge our sandbox component seamlessly into an app, using the APK rewriter we developed.

Specifically, we use apktool to decompile an Android application package file (APK) into human-readable smali codes, include our sandbox component and then recompile the files back into an executable APK. This is difficult because each APK is an integral structure, and including our sandbox component is not as easy as copying all the files — we have to make sure each component serves its designed functionality at the right place, and this can be challenging especially when we need to consider the integration of UI. This brings the following technical challenges:


Identifiable:

U
Device ID: FCC2-04CF-1C44-7BC6
Last Used: never used before

Anonymous:

?
Device ID: ????-????-????-????
Last Used: never used before

Pseudonymous: OFF

1
Device ID: 3BC9-C531-A98F-978B
Last Used: 2013-11-07 23:04:12

2
Device ID: 7442-D32F-D58C-B301
Last Used: 2013-11-07 23:01:02

3
Device ID: D226-53B1-9E49-B556
Last Used: 2013-11-07 23:04:57

+
Device ID: XXXX-XXXX-XXXXX
Last Used: to be determined

Fig. 5. The UI design of Mask.

1) Sandbox Initialization: The APK rewriter needs to make sure that the sandbox component will be initialized before any component of the original application executes. This is achieved by exploiting an application base class which is provided by Android to maintain global application states. The nice property of this base class is that it is the first user-controlled component that gets initialized for any process of the app. Our APK rewriter will go through the app's existing codes and checks whether the application base class already exists. If exists, we modify the existing application base and make it a subclass of the application base defined by us; otherwise, we directly insert our application base. The sandbox initialization logic is programmed into the static code section of this application base class and is guaranteed to be the first to execute.

2) UI Integration: All UI elements integrated in an app are referenced with a universal unique id, indexed under *res/values/ids.xml* and *res/values/public.xml*. To integrate new UI elements into an existing app, our APK rewriter automatically tracks and assigns the empty slots within the existing ids. Moreover, the APK rewriter needs to ensure control will be returned to the app after our UI cuts in line. It works by going through the manifest file, identifying the app launcher activity and statically writing an initialization logic for the launcher activity into the `onDestroy` functions in our UI activity.

Finally, the APK rewriter also parses the manifest file to get the list of processes this app hosts. This information will be hard-coded into the smali codes of the sandbox manager to enable Mask's support for multiple processes.

TABLE III
UI & Sandbox Management Overhead

Category	*Response Time*
Load UI	169.2 ms
Create Sandbox	68.7 ms
Run Sandbox	382.2 ms
Destroy Sandbox	26.8 ms

TABLE IV
Mobile App Runtime Overhead

Category	*Bench*	*Unit*	*Overhead (%)*
File	Seq Read	MB/s	$< 1\%$
	Seq Write	MB/s	1.3%
	Rand Read	MB/s	$< 1\%$
	Rand Write	MB/s	$< 1\%$
	Create	TPS	2.1%
	Delete	TPS	4.7%
Database	Insert	TPS	2.3%
	Update	TPS	9.0%
	Delete	TPS	3.4%
IPC	Filter	TPS	$< 1\%$
	Reformat	TPS	37.8%

TPS: Transactions Per Second

D. Performance Overhead

Mask incurs two types of performance overhead: on sandbox management when a sandbox is created, destroyed or executed; and during application runtime, after an app starts execution in a sandbox of the user's choice.

The overheads on sandbox management are measured by instrumenting a testing app with Mask and timers. Then we perform selected sandbox management tasks and log the output of these timers. As the results in Table III show, the most time-consuming actions in sandbox management are loading UI and running a sandbox, each taking a few hundred milliseconds. However, since these actions happen only once during a *session*, the cumulative overhead on sandbox management is still minor—far less than one second.

The overhead on apps' runtime originates from Mask's user-level sandbox component when it intercepts inter-process communications (IPCs) and file system operations. To measure the overhead incurred by redirecting file system operations, we choose a benchmark app, AndroBench [12], which is designed for measuring storage performance on Android devices. Besides the test benches included in AndroBench, we added two more tests to measure the performance of creating and deleting files. Each test bench is executed 10 times with Mask enabled and disabled. To evaluate the overhead caused by intercepting IPCs, we use a synthesized benchmark which contains two test benches, measuring the overheads incurred by IPCs filtering and reformatting, respectively. IPC filtering differentiates the IPCs we are interested in, such as getting device ID, from those we are not, such as getting location updates, while IPC reformatting reconstructs a low-level binary sequence into high-level objects and modifies the corresponding persistent identifiers. The IPC filtering incurs overhead to any IPC between an app and other parties, while the IPC reformatting overhead is incurred only for those IPCs that return personal or device identifying information to the app.

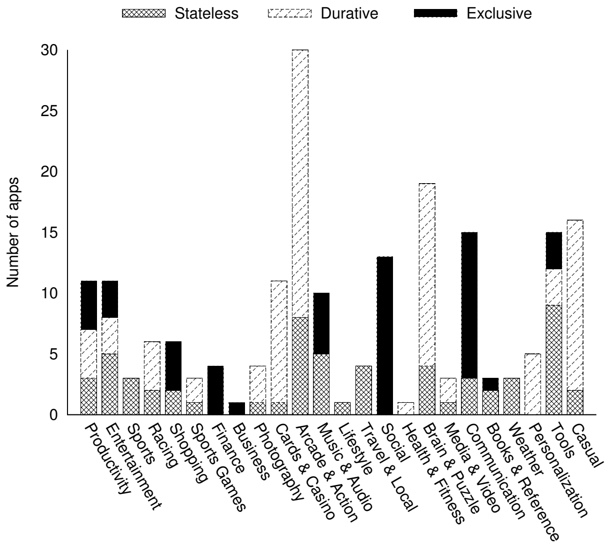


Fig. 6. Breakdown of apps according to their usage patterns on a per-category basis

The results on application runtime overheads are summarized in Table IV. The performance degradation on file system operations is minor because the only overhead incurred is for transforming the paths in the app's original storage to the paths in the sandbox's shadow storage. This transformation is much lighter-weighted compared to file system IOs. We also found the IPC filtering overhead to be negligible, meaning that Mask does not affect the performance of most 'uninteresting' IPC calls. By contrast, the IPC reformatting overhead is significant (more than 37%) because parsing byte array into high-level objects, for example java objects, is expensive. However, since we only reformat a very small portion of all the IPCs—only those return persistent identifiers—the overall performance degradation is found negligible.

E. Applicability

Mask applies different *private execution modes* for mobile applications with different usage patterns (exclusive, durative or stateless). Here, we case studied the top 200 free apps in Google Play and study how many of them can be used with Mask. We assume a privacy-aware user who always selects the usage pattern that delivers the highest privacy level without compromising the major functionality of the app. We classify the apps according to the usage patterns and further break down the numbers into each functional category, as shown in Fig. 6. To sum up, 29% of the apps can be used statelessly, 43% can are used duratively and 25% have to be used exclusively. To note, Mask is only applicable to 97% of all the apps, excluding apps designed for system usage, such as file explorer, anti-virus software, etc. Sandboxing these apps fundamentally violates their functionalities and results in unpredictable results or even crashes.

We also conducted a study on 27 mobile users who had neither prior knowledge of the Mask project nor domain-specific expertise. The users are recruited by posting surveys in our university library. Our findings are summarized as follows:

- 80% of the mobile users are aware that app developers or advertising libraries may conduct user profiling. 40% of the mobile users consider this as a moderate or serious privacy threat, 45% as a minor privacy threat while 15% of them are not concerned with it at all.
- 60% of the mobile users believe maintaining multiple isolated profiles for the same mobile app will provide better privacy protection. Of these people, more than 70% are willing to take actions if theyre given such an option.

V. Related Work

There have been numerous studies on information access control on smartphones [1–5]. They focus on detecting and defending illegitimate collection of sensitive user information, by malicious mobile apps or third parties. Orthogonal to these studies, Mask focuses on *unregulated aggregation* of sensitive user information, irrespective of how it is collected. There also exist other efforts on the issue of information aggregation in the mobile ecosystem [7, 13–16]. Most of them only target a restricted scenario — advertising agencies — and assume mobile apps are trustworthy. Different from these works, Mask considers a more general and realistic scenario and and breaks unregulated aggregation by both mobile apps, advertising agencies and network sniffers. Linkdroid [16] tries to solve a similar problem but requires extensive modifications on the smartphone OS. Mask, instead, utilize the usage patterns of apps and strikes a useful trade-off by delivering private execution modes in the user-level.

MoRePriv [17], πBox [18] also focused on resolving the conflict between privacy and personalization of mobile apps. MoRePriv argued that personalization should be provided by the OS instead of apps. Similarly in principle, πBox shifts the responsibility of protecting privacy from the app and its users to the platform itself. By contrast, Mask neither advocates a new ecosystem, nor requires modification to the existing one. It provides a practically deployable design which allows end-users to 'negotiate' with the mobile app at the client side.

VI. Conclusion

In the current mobile app ecosystem, app usages of a user are linkable by default. This allows an interested/curious party to conduct unsolicited user profiling, targeted advertising or public surveillance. In this paper, we designed and implemented a practical solution called Mask, allowing users to unlink app usages that are functionally independent of each other. Specifically, we introduce a set of *private execution modes* and give users options to specify whether and within which scope his current app session should be linkable. We presented the technical details and challenges of our user-level implementation, and evaluated the performance and applicability of Mask.

Acknowledgement

The work reported in this paper was supported in part by the US National Science Foundation under Grant CNS-1505785.

References

[1] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're looking for: retrofitting android to protect data from imperious applications," in *Proceedings of the 18th ACM conference on Computer and Communications Security*. ACM, 2011, pp. 639–652.

[2] M. Nauman, S. Khan, and X. Zhang, "Apex: extending android permission model and enforcement with user-defined runtime constraints," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*. ACM, 2010, pp. 328–332.

[3] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones." in *OSDI*, vol. 10, 2010, pp. 255–270.

[4] S. Bugiel, S. Heuser, and A.-R. Sadeghi, "Flexible and fine-grained mandatory access control on android for diverse security and privacy policies," in *Presented as part of the 22nd USENIX Security Symposium*. Berkeley, CA: USENIX, 2013, pp. 131–146. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/papers/bugiel

[5] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "Pios: Detecting privacy leaks in ios applications." in *NDSS*, 2011.

[6] N. Xia, H. H. Song, Y. Liao, M. Iliofotou, A. Nucci, Z.-L. Zhang, and A. Kuzmanovic, "Mosaic: quantifying privacy leakage in mobile networks," in *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*. ACM, 2013, pp. 279–290.

[7] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Investigating user privacy in android ad libraries," *IEEE Mobile Security Technologies (MoST)*, 2012.

[8] I. Gartner. (2013) Forecast: Mobile app stores, worldwide, 2013 update. [Online]. Available: http://www.gartner.com/DisplayDocument?id=2584918

[9] S. Tambe, N. Vedagiri, N. Abbas, and J. E. Cook, "Ddl: Extending dynamic linking for program customization, analysis, and evolution," in *In Proc. International Conference on Software Maintenance*, 2005.

[10] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical policy enforcement for android applications," in *Proceedings of the 21st USENIX conference on Security symposium*. USENIX Association, 2012, pp. 27–27.

[11] M. Balakrishnan, I. Mohomed, and V. Ramasubramanian, "Where's that phone?: geolocating ip addresses on 3g networks," in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*. ACM, 2009, pp. 294–300.

[12] C. S. L. at Sungkyunkwan University. (2012) Androbench. [Online]. Available: http://www.androbench.org/wiki/AndroBench

[13] S. Han, J. Jung, and D. Wetherall, "A study of third-party tracking by mobile apps in the wild," UW-CSE, Tech. Rep., 2011.

[14] S. Shekhar, M. Dietz, and D. S. Wallach, "Adsplit: separating smartphone advertising from applications," in *Proceedings of the 21st USENIX conference on Security symposium*. USENIX Association, 2012, pp. 28–28.

[15] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, "Addroid: Privilege separation for applications and advertisers in android," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*. ACM, 2012, pp. 71–72.

[16] H. Feng, K. Fawaz, and K. G. Shin, "Linkdroid: reducing unregulated aggregation of app usage behaviors," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 769–783.

[17] D. Davidson and B. Livshits, "Morepriv: Mobile os support for application personalization and privacy," MSR-TR, Tech. Rep., 2012.

[18] S. Lee, E. L. Wong, D. Goel, M. Dahlin, and V. Shmatikov, "πbox: a platform for privacy-preserving apps," in *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2013, pp. 501–514.