Privacy Capsules: Preventing information leaks by mobile apps

Raul Herbster
MPI-SWS

Scott DellaTorre
University of Maryland

Peter Druschel
MPI-SWS

Bobby Bhattacharjee
University of Maryland

Abstract

Preventing the leakage of user information via untrusted third-party apps is a key challenge in mobile privacy. We propose and evaluate *privacy capsules (PCs)*, a platform execution model for mobile apps that prevents the flow of private information to untrusted parties by design. With PCs, apps execute in two sequential phases. In the *unsealed* phase, the app has no access to sensitive input but full access to untrusted network resources. In the *sealed* state, the untrusted app has access to sensitive input, but can no longer communicate with untrusted resources. Privacy capsules are implemented by the mobile platform, are language independent, and require few changes to apps. Using a prototype PC implementation in Android, we show that PCs have low performance and energy overhead, and are suitable for a large class of apps.

1. INTRODUCTION

Mobile apps on major platforms provide a dizzying array of functionality, and form the basis for the success of these platforms. Users buy relatively inexpensive, often free, apps that cater to the necessities (schedule, online access, health monitoring) to entertainment (games, music) and beyond. The spectrum of functionality available on these devices also make them a repository of sensitive personal information, which is either explicitly added by the user (credit cards, passwords) or inferred from sensor readings (location, activity), and is often available to downloaded apps.

Prior work has shown that many third-party apps access information for which they do not have a legitimate need; some forward information to providers without the user's knowledge or consent. Egregious examples, such as flash light apps that access and upload user's location, phone number and contacts, are relatively easy to flag as malware [4] and have been the subject of popular press articles [6]. However, the problem is typically more subtle: Felt et al. [20] found that one third of 940 apps on the Android Market requested more permissions than needed. Enck et al. [19] analysed 30 popular Android applications and found that half of them send users' location to advertising servers, and two thirds handle private data in a suspicious manner.

Beyond malware, even benign apps that are poorly designed or have bugs pose a threat to user privacy. A typical class includes apps that use insecure network protocols for service requests: according to one source, 92% of the top 500 Android applications available in app stores (Amazon and Google Play) use insecure protocols and/or leak sensitive information in other ways [3]. Documented bugs [7] have leaked plain text chat logs and passwords for hundreds of millions of users.

There are two primary techniques for restricting app behavior. The first, which is widely deployed, is for the app to pre-declare the set of resources it requires. Resource access is often coupled with user consent at the time of first use. In practice, this form of coarse-grained access control has proven to be ineffective. Most users are unsure whether an app legitimatately requires a particular resource; when faced with a choice to either approve access or forego the use of the app, they tend to approve [21, 23]. Further, the temporal granularity over which access is granted is coarse-grained, e.g., once access to location is granted, an app can query present location anytime until the access is explicitly revoked. Finally, even if an app must access a set of resources, there is no way to prevent unwanted information flow among those resources; for instance, if an app requires access to current location and network, platform access control cannot prevent the app from forwarding the current location to any network peer.

The second approach, still within the research domain, seeks to address app misbehavior using more fine-grained access control [26], or information flow control (IFC) [9, 19]. Fine-grained access control can minimize or prevent illegitimate resource accesses, but cannot detect or prevent illegitimate flows. Solutions relying on IFC can detect and prevent illegitimate flows, but have high performance and energy overhead, and suffer from false positives [25].

We propose **Privacy capsules (PC)**: a low-overhead technique that provides containment of sensitive information. By design, the containment properties of PCs are trivially simple to prove. PCs' run-time efficiency and containment guarantees come at a cost: PCs require applications to be structured in a particular manner, and may preclude certain classes of applications.

PCs are implemented as part of the mobile platform and enforce a two-phase execution model:

- An app's execution starts in the *unsealed state*, where

MobiSys '16, June 25–30, 2016, Singapore.

ACM ISBN 978-1-4503-4269-8/16/06..

DOI: http://dx.doi.org/10.1145/2906388.2906409

it can freely access untrusted resources but has no access to private information.

- Once the app explicitly chooses to switch to the *sealed state*, it can access private information but can no longer access untrusted resources. The app cannot return to the unsealed state, but only terminate and restart (in the unsealed state).

A platform policy specifies what information is considered private. By construction, the PC execution model prevents the flow of private data to untrusted channels, i.e., it is impossible for the app to leak private information[1].

PC apps must be written to conform to this execution model. We believe this approach is viable because for many classes of applications, especially those that implement a service-query model, the required changes are minor, especially for newly developed apps. The containment guarantees provided by the PC model provides adoption incentives to both platform providers, app developers, and users. The platform will guarantee that apps, no matter how poorly designed or malicious, cannot leak private information; app developers do not have to secure their back end storage or audit their designs against attacks or misconfiguration, and users no longer have to trust apps, or debate privacy and access settings.

To enable rich functionality without compromising privacy, PCs provide additional mechanisms to enable legitimate flows of private information: sealed repositories, sealed channels, and selective declassification.

- Using a *sealed repository (SR)*, an app in sealed state can store private information persistently across executions. Data stored in an SR is encrypted with a user- and app-specific key that is never revealed by the platform, and therefore cannot be leaked by the app. An SR can reside in untrusted storage, such as the provider's Cloud.

- Using a *sealed channel (SC)*, live voice/video/chat data can be exchanged among instances of an app in sealed state on different users' devices. App instances and participating users are authenticated, and information in the channel is encrypted with keys that are never revealed to the app.

- Finally, apps may effect legitimate flows of private information to third parties using *selective declassification (SD)*. With SD, specific information can be disclosed to a specific third party with the user's express consent.

Using these mechanisms, we describe the design of three prototype PC-compliant apps, each of which represent a class of apps. A *train scheduling and ticketing* app represents apps that serve maps, event/transportation schedules, and product catalogs, and allow users to browse/purchase items. A *wellness* app represents the class of apps that monitor health and fitness sensors, analyze and tabulate data, and enable the sharing of selected data with friends, coaches, or health care providers. Finally, we present a *live chat* app, which represents the class of apps focused on communication with other users.

The rest of this paper is organized as follows. We discuss background and related work in Section 2. Section 3 presents the design of privacy capsules. Section 4 presents guidelines for developing PC-compliant apps and sketches our PC-compliant sample apps. An experimental evaluation follows in Section 5, and we conclude in Section 6.

2. BACKGROUND AND RELATED WORK

Personal mobile devices can capture a continuous record of their users' location, activity, and audiovisual environment, and are commonly used to store schedules, passwords, banking records, and other sensitive information. The primary mechanism for ensuring user privacy and security is an audit by the platform provider, which usually ensures that gross malware is culled from the app marketplace.

Unfortunately, privacy leaks via apps are common and come in three flavors: *Incidental* privacy leaks occur as a side effect of the app's operation. For instance, an app that queries a database on a provider's site for information related to the user's current location or interests leaks information to the provider as part of each query. *Accidental* privacy leaks occur due to app bugs and vulnerabilities that expose the user's private information to network eavesdroppers, attackers, or other apps. Finally, a *malicious* leak occurs when an app deliberately and needlessly forwards private information to third parties. All three types of privacy leaks are amplified when the app requests and is granted access to private information that is not required to implement its documented functionality.

In this section, we survey the commonly used mechanisms, both deployed and in research, that have been developed to address privacy leaks.

2.1 Mobile platform permission systems

Mobile platforms including Android, iOS, and Windows Phone control the resources each third-party app can access. Users are asked to grant an app access to resources like the network, camera, microphone, location provider, calendar, contacts, etc., either at installation time or upon first access. Prior work has shown that users tend to approve most requests, presumably because they are unable to judge whether an app has a legitimate need for a resource, and may not be fully aware of the risks [21].

Several prior works aim to improve the permission system found in mobile platforms. Stowaway [20] analyzes an app statically to determine the set of resources it actually needs from its API usage. Bartel et al. [14], Au et al. [11] and Atzeni et al. [10] propose similar approaches. Dr. Android [26] provides more fine-grained access control for Android, for instance, by distinguishing read and write access to a resource, but still relies on correct decisions taken by lay users.

AppGuard [12], Aurasium [36], and NativeGuard [34] rely on inlined reference monitors to enforce more fine-grained access control policies than the platform. In these solutions, accesses to resources are dynamically monitored and the policies enforced. The user is notified whenever a certain application does not follow policy. In AppGuard, the policies are defined by the user. Aurasium monitors the calls between the app and the OS. It defines a fixed set of policies; for instance, calls to functions like `ioctl()`, `open()`, `fork()`

[1]Our prototype does not account for timing attacks, but such attacks could be mitigated using randomization and other well-known techniques.

and `dlopen()` are considered as potentially risky. NativeGuard verifies if the native libraries follow the permission initially defined by the developer as part of the manifest file.

2.2 Information flow control

Information flow control (IFC) [32] has been used to enforce data confidentiality and integrity in different environments [15, 16, 18, 37], including mobile [19, 24, 30]. Unlike access control, IFC can track the flow of private information through an app. For instance, IFC can prevent an app from sending the user's private information to the network, even though it has access to both resources.

TaintDroid [19] uses fine-grained taint tracking to detect private data leakage in Android apps. It is a considerable advance over Android's permission control, but does not consider implicit/indirect flows, which arise from the program's control flow [27]. Moreover, TaintDroid incurs considerable performance and energy overheads. Some forms of IFC can be enforced using static analysis([30], [24]); while efficient, these tools can identify a relatively small class of information leakage, and are constrained to apps where the source code (and all libraries) are available for analysis.

SpanDex [17] relies on runtime techniques to mitigate leaks of specific private information like passwords through implicits flows in mobile apps. ReCon [31] uses machine learning to identify personally identifiable information (PII) in network traffic, enabling users to detect and manage potential privacy leaks by mobile apps. ReCon must be trained to identify specific PII and is subject to classification accuracy. The system can be deployed either as part of the mobile platform, or as a trusted middlebox with access to cleartext network traffic.

2.3 Sandboxing approaches

Sandboxing is a well-known technique for isolating the execution of untrusted programs, and can be used to control app access to sensitive information and resources. Sandboxing has been proposed for social networks [33, 35] and for mobile platforms [29].

Lee et al. [29] propose πBox, a sandbox technique for Android where apps can establish connections outside the sandbox through restricted storage and communication channels, preventing apps from exposing data via world-readable files. This approach does not prevent the application from accessing sensitive data from sensors, or from leaking it to the application provider.

In contrast to prior work, PCs enforce a simple execution model for untrusted apps, which rules out privacy leaks by design. Enforcing the PC execution model does not require expensive runtime intervention like taint tracking. Instead, PCs intercept untrusted apps' I/O operations and modify them according to the rules for the app's current sealing state.

3. DESIGN

Privacy capsules are implemented by a trusted mobile platform. The platform includes the OS, and a set of basic services, such as a location provider, a Contacts application, and a GUI, all of which are also assumed to be trusted.

Execution model: The PC execution model is depicted in Figure 1. The execution of an app starts in the unsealed state, where it has access to public inputs and untrusted resources. Later in its execution, the app process may switch to sealed state, where it has access to private inputs and trusted resources, but can no longer write to untrusted channels. Once in sealed state, the process cannot return to unsealed state, only terminate.

All sensor data (health/fitness, inertial, camera, microphone), geolocation, GUI inputs, phone status, databases (e.g,. contacts, calendars) are private by default. Public files, network connections, and interprocess communication (IPC) with untrusted processes in unsealed state are considered untrusted channels.

The main components of PCs are the following:

Key manager: The PC key manager maintains, for each app and user, a set of cryptographic keys used to encrypt data in sealed repositories and channels. Apps can request keys be created and select keys for use with a specific repository or channel. Apps refer to keys using handles, but cannot access the keys.

Sealed respositories (SR): In sealed mode, an app process may read and write an SR, allowing the app to persistently store data across executions. Data stored in an SR is encrypted with a user-specific key maintained by the PC key manager. SRs may reside in untrusted storage, like the Cloud.

Sealed channels (SC): An SC allows an app process in sealed mode to communicate with instances of the same app executing on other users' devices. The communicating apps are authenticated by the PC platforms, and communication is encrypted with a channel-specific key. Apps using a SC retain access to GUI events, and the device's microphone and camera. SCs are designed to enable person-to-person communication apps.

Selective declassification: An app process in sealed mode can disclose a particular data item to a particular third party with the user's express consent. To allow the user to determine the nature of the information disclosed, PCs track the type and provenance of all input data. When an app asks for selective declassification, the PC platform presents the user with the origin and type of the information to be disclosed and asks for confirmation.

The PC execution model ensures that no private data can flow to untrusted channels or parties, except via selective declassification.

PC-compliant apps can be installed and executed alongside ordinary apps on the same platform. We envision that users may continue to use ordinary apps from their bank, employer, or other reputable providers they trust. However, privacy conscious users would insist on PC-compliance for apps from less well-known providers, and for apps that require access to sensitive information (health, financial, private communication, etc.)

3.1 Threat model

The mobile platform, including the PC implementation, the OS, and platform apps, are trusted[2]. Attacks against these components are out of scope. We believe these assumptions are reasonable because (i) reputable platform providers

[2]When sealed channels are used, the mobile platforms on all communicating devices are trusted.

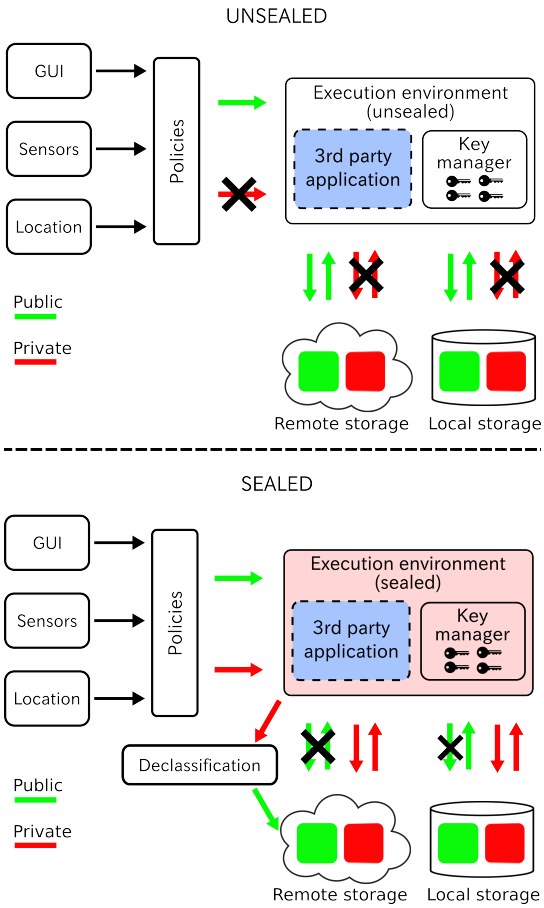

Figure 1: Privacy Capsules execution model

	App State	
	Unsealed	Sealed
Platform, Sys. Events		
App launch/fg/bg events	allow	allow
GUI events	deny	allow
All other sys. events	deny	allow
Device Sensors		
Sensor	deny	allow
Display	allow	allow
Audio	deny	allow
Microphone	deny	allow
Camera	deny	allow
Precise location	deny	allow
Public location	allow	allow
Device/network status	deny	allow
System Services		
Calendar read/write	deny	allow
Contacts read/write	deny	allow
Phone calls/SMS	deny	deny
Files/network		
Open/Close streams	allow	deny
Read streams	allow	allow
Write streams	allow	deny
PC Key Management		
Create/set keys	allow	deny
Cryptographic key access	deny	deny
PC sealed repository/channel		
SC/SR creation	allow	deny
SR/SC read/write	deny	allow

Table 1: Default PC policy for apps in sealed and unsealed states

have an interest in protecting users' privacy and the integrity of their platform, and (ii) while technically sophisticated users may compromise ("jail-break") their platform, compromising the integrity of PCs will only place their own privacy at risk.

By design, the guarantees provided by PCs hold regardless of app bugs, vulnerabilities, or deliberate misbehavior. For this work, it is assumed that apps do not exploit timing channels to leak private information. We believe that timing channels can be thwarted by shaping app I/O operations in the PC platform. However, a closer analysis and prevention of timing channels in the context of PCs remains the subject of future work.

3.2 State-dependent access restrictions

PCs prevent illicit information flows by mediating I/O operations depending on app's current state. Table 1 lists relevant operations and their state-dependent restrictions.

In the unsealed state, an app process has full access to untrusted channels like network connections and files, and the public location as defined by the location policy (see Section 3.4). The process can open but not read or write sealed repositories, sealed channels, and sources of private data, including sensors and GUI events. The process can interact with other processes in unsealed state only, and may receive a limited set of system event types that are not privacy sensitive. Finally, the process can instruct the PC key manager to create keys, and to select keys for use with specific sealed repositories and channels.

In the sealed state, the process has access to all private inputs, system events, databases, and status information, as well as sealed repositories/channels, but it can no longer write to untrusted channels. The process can interact with other processes in sealed state only. To prevent an app from leaking private information, the process can no longer open new files, network connections, or input devices, create nor select keys. These events can be observed by untrusted parties and could be exploited to encode private information as part of a covert channel (see Section 3.5).

Access to APIs for cellular voice calls or simple text messages (SMS) is prohibited in sealed mode, because it is difficult to effectively prevent leakage of private information via these technologies. However, PC-compliant apps for VoIP or secure messaging (chat) can be implemented, as shown in Section 4.5.

3.3 App launch

When a user or an app in unsealed state launches a PC-compliant app, the new app process starts in unsealed state. The new process can switch to sealed state when ready.

In Android, apps can launch other apps through events called *Intents*, which may pass arguments from the launching

app to the launched app. Therefore, when an app process in sealed state launches an app, PC needs to ensure that no private information is leaked. In general, an app in sealed state may launch another app only if the launched app is a trusted platform app, the launched app is PC-compliant and forcibly started in sealed state, or the arguments passed in the intent are selectively declassified.

3.4 Location policy

Privacy capsules require no app-specific policies, and no user-specific policies with one exception: A user may wish to select a location privacy policy. The device's current location as defined by latitude/longitude is considered private, like any other sensor input. However, the location policy may define a *public geofence*, which is available to apps in unsealed state. The public geofence is the current location expressed as a geographic region at a particular political/administrative level. The top levels are world, continent, country. For a location in the US, the lower levels are state, county or independent city, municipality, ZIP code, street, address. An example location policy might say that the current location at the state level is public, and can be accessed by apps in unsealed state.

PCs automatically maintain a public geofence according to the location policy in effect. When the device moves out of the region covered by the current public geofence, PCs conservatively switch to the next higher level (e.g., from state to country). The user is notified of the change and has the option to adjust the public geofence manually. To avoid leaking information about the user's current location (i.e., somewhere at the geofence boundary) to unsealed apps, automatic changes to the public geofence are delayed randomly.

Apps in unsealed state may invoke a trusted map application to request the user to specify a location, e.g., a destination to be used for a navigation or travel booking application. The PC platform makes this input available to the app after generalizing the location to the same political/administrative level as the current public geofence.

3.5 Sealed repositories (SR)

Sealed repositories allow apps to store private information (profiles, logs, keys, accumulated sensor information) across launches. An SR can be stored as a local file on the device, or on a Cloud provider via HTTPS. Data stored in an SR is encrypted with a cryptographic key created and stored on behalf of the app by the PC key manager. The key itself is never revealed to the app; as a result, the app can read and write an SR whenever it is in sealed mode, but it cannot reveal its content to a third party.

An SR must be created and opened while the app is in unsealed mode. Creating or opening binds the name of a file-backed SR, and the URL and HTTPS GET/POST arguments for a Cloud-backed SR. Requiring creation and opening in unsealed mode ensures that no private information can be encoded in the set of SRs, their names, URLs, or GET/POST arguments, which can be observed by third parties; however, apps may still negotiate app-specific authentication without involving the platform. Once in sealed mode, the platform uses a well-known API (GET/POST/REST) to communicate over the channel created by the app.

SRs must be read sequentially, in their entirety, and only once per execution to prevent an app from encoding private information in its pattern of access.

3.6 Sealed channels (SC)

Sealed channels enable communication among instances of an app that execute on different users' devices, and support typical communication apps like messaging, chat, video calls, or multiuser games. When combined with appropriate key management as in Persona [13], SC can even be used to support private online social networks.

An SC must be set up in unsealed mode. When an app requests an SC, the PC platform launches the contacts app to have the user select participants. (Since the app is in unsealed state, it has no access to the contents of the contacts database). Using the public keys in the contacts database, the platform then authenticates the selected participants and establishes a shared session key. When the app instances on the participating devices switch to sealed state, they can access the SC.

Note that sealed mode app instances can freely share information. This is necessary as the participants wish to exchange video, audio, text typed on the keyboard, or other GUI events. To prevent such app instances from sharing other information like user's sensor inputs, locations, databases (e.g., calendar, contacts), or sensitive device state, an app with an active SC is denied access to private inputs other than GUI events, camera, and microphone.

3.7 Selective declassification

In some cases, users wish to deliberately disclose some private information to a particular third party. For instance, a user of a purchasing app who wishes to purchase an item needs to disclose to the vendor their choice of item, typically along with a name, shipping address, or payment information. As another example, a user of a fitness/health app wishes to disclose some of her sensor data to a coach or health care provider.

Selective declassification allows an app in sealed mode to declassify information with the express consent of the user. For this purpose, the trusted platform requires the user to confirm the declassification of a specific data item to a specific third party.

Users must be able to verify what information they are asked to declassify. For this purpose, the PC platform signs all private input data consumed by PC-compliant apps with its provenance metadata. Sensor data is signed with its type, sensor, and time of capture. GUI input is signed with the GUI mask and time at which it was entered.

Apps are allowed to declassify only information that corresponds directly to a set of signed input data items. This restriction ensures that an app cannot trick the user by encoding additional, private information in the data to be declassified, e.g., using steganography[3].

Finally, when an app requests declassification, the platform displays the metadata of the information to be declassified, along with the specific third party to which the app wishes to declassify. The third party may be a provider

[3]We considered allowing declassification of arbitrary data by presenting the user with the union of the metadata of all inputs consumed by the app. This approach is safe in terms of information flow because the metadata covers all inputs that could be reflected in the to-be-declassified data. However, in general the union of the metadata lacks specificy, which may make it difficult for users to make informed decisions about declassification.

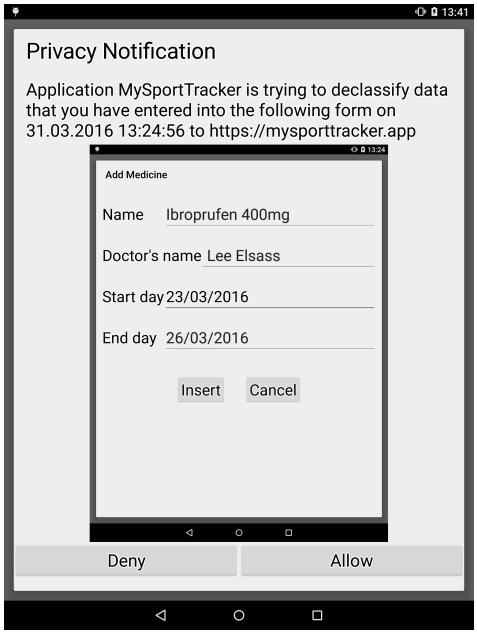

Figure 2: Example disclosure dialogue

identified by an authenticated HTTPS domain (via a secure connection), a specific person identified by a public key (via a file encrypted with that public key), or another app (via an intent). Only if the user confirms is the information declassified and encrypted for the third party.

Figure 2 shows an example declassification confirmation dialog for information about medication that was previously entered by the user. In the case of sensor information, the displayed metadata identifies the sensor, the resolution, and the period covered by the data.

Unlike existing permission systems, selective declassification enables users to decide on a case-by-case basis whether specific information identified by its metadata should be disclosed to a specific third party. The app remains untrusted, but the user implicitly entrusts the third party with the declassified information.

Generalized Declassification.

Both Sealed Channels and Selective Declassification are examples of declassification whereby users' input is required to explicitly allow specific information to be declassified to a known destination. More generally, we envision a declassification architecture for PCs that uses a standalone key distribution infrastructure to disseminate Attribute-based keys corresponding to device resources (similar to the key management in [13]). Sealed channels and selective declassification allow us to demonstrate the utility of PCs and implement useful classes of applications; generalized declassification remains a topic of future work.

4. DEVELOPING APPS WITHIN PCS

To be able to run inside a privacy capsule and benefit from its privacy guarantees, apps must conform to the PC execution model. In this section, we present the PC API, and discuss guidelines for developers who wish to write PC-compliant apps. We also describe three PC-compliant apps we implemented: a train timetable app, a health/fitness app, and a simple text chat.

4.1 Privacy capsules API

The privacy capsules API is summarized in Table 2. The API calls are grouped into three categories. The basic API provides operations for creating and selecting keys, for creating and opening file-based SRs, and for swiching to the sealed state. The HTTP connection API provides means for creating and opening HTTP-based SRs, and for declassification into HTTP connections. The secure channels (SC) API provides means to establish a sealed channel (SC). With the exeption of the SC API, which is currently available only in Java, APIs are also available for C programs.

4.2 Guidelines for developing PC apps

Next, we give general guidelines for developing PC-compliant apps. A key question for the developer of a PC-compliant app is how to split the apps's functionality between the unsealed and sealed states. At a high level, functions that require communication with untrusted third parties require unsealed state, while functions that depend on private data require sealed state. Moreover, the two states must occur sequentially and only one transition is allowed, from unsealed to sealed state. Depending on the nature of an app, this constraint, which ensures that no private information can flow to untrusted channels, may require some care. Following, we discuss common patterns of app information flow, and how to accomplish them in a PC-compliant manner.

Maintaining private information across launches: Many apps store private information like the user's search and activity history, personal profile, authentication credentials, etc., persistently across app launches. In many cases, the information is stored in the Cloud on the app provider's site. PC-compliant apps can store such information in a sealed repository, accessed from sealed state.

Sensitive database queries: A large class of apps provide maps and directions, transportation scheduling and ticketing, discovery of nearby resources, shopping, and may include advertising components. Such apps typically query a Cloud database, where the queries depend on private data like the user's location, destination, history, profile, and interests. Privacy-preserving, PC-compliant versions of such apps maintain a cache of the database on the local device, and resolve queries locally from the cache. In unsealed state, the app updates its cache from the Cloud, along with new tips, advertisements, or promotions pushed by the Cloud provider. Once in sealed state, the app resolves user queries and performs ad selection from the local cache.

Purchase transactions: Apps that perform reservation or purchase transactions must necessarily disclose some private information, like the product ID, name, shipping address, or payment details, to the vendor. PC-compliant apps use selective declassification to disclose such specific information to a particular vendor from the sealed state, with the user's express consent.

Health/Fitness/Nutrition advice: Apps that allow users to seek advice from health care providers, fitness coaches, or nutrition advisers necessarily release private information (e.g., health and fitness sensory data, nutrition/activity logs) to third parties. PC-compliant apps use

Seal/Unseal API

Function	Description	Java (class `PrivacyCapsulesService`)
Seal	Seals the app	`boolean seal()`
Get seal status	Checks if app is sealed	`boolean isSealed()`
Key creation	Creates a key and returns a handle	`long createKey()`
Set key	Associates a key with the given SR file descriptor	`long setKey(FileDescriptor, long)`
Associate handle with name	Associates a handle (long) with a textual name	`boolean setHandlePair(long,byte[])`
Retrieve handle by name	Retrieves handle associated with given name	`long retrieveHandlePair(byte[])`

HTTP Connection

Function	Description	Java (package `java.net.*`)
Open connection	Opens a HTTP connection with a remote SR	`HttpURLConnection URL#openConnection(long)`
Open connection for data disclosure	Opens a PC HTTP connection for selective data disclosure	`HttpURLConnection URL#openDisclosureConnection(long)`

Secure Channels

Function	Description	Java (class `PrivacyCapsulesService`)
Open SC	Shows a dialog with existing contacts; opens a SC and returns a file descriptor associated with each selected contact.	`ParcelFileDescriptor[] createSecureChannel()`
Get SC input stream	Returns the input stream associated with a SC	`getSecureChannel[Client/Server]Input(ParcelFileDescriptor)`
Get SC output stream	Returns the output stream associated with a SC	`getSecureChannel[Client/Server]Output(ParcelFileDescriptor)`

Table 2: Summary of privacy capsules API

selective declassification to disclose such specific information to a particular provider from the sealed state, with the user's express consent.

Live communication with other users: Communication applications like voice/video calling and chat necessarily require live communication with the participants of the call. PC-compliant apps set up a sealed channel in unsealed mode, and use the channel from a restricted sealed state that grants access to GUI events, camera, and microphone.

Social networking: Many apps have a social networking component, where private information is made available selectively to specific groups of other users. While we have not yet experimented with such apps, we believe that a generalized form of declassification based on attribute-based key management can support such apps.

Sensor matching: Apps like "bump" [2, 5] require accelerometer, location and clock readings to be compared between different devices. In our prototype, such apps can be implemented by selectively declassifying the sensor inputs to a Cloud-based matching service. In future work, we plan to add support for privacy-preserving sensor matching using multiparty secure computation, as in [8].

Next, we describe three PC-compliant apps we developed. Each sample app represents a large class of similar apps. Designing and implementing these apps did not present any difficult challenges related to the PC execution model. With the PC execution flow in mind, it was possible to quickly design the apps.

4.3 DB train timetable app

We implemented a train timetable app that replicates the functionality of a proprietary app offered by the German railway company DB (Deutsche Bahn). Our app relies on DB's Cloud backend service, which offers an open Web API.

In normal operation, the DB app queries the DB backend and returns a travel plan to the user, which reveals the user's point of departure (often the current location), destination and time of travel to the DB backend service. Users also have a ticketing option, which submits the user's name and payment details to the DB backend as part of a transaction, and returns an online ticket. The DB app supports an offline mode, where the timetable is cached on the device and the app resolves information queries from the cache.

Our PC-compliant version of the app instead follows the execution flow shown in Figure 3. In the unsealed state, the app updates its local cache of the timetable and realtime information from the DB backend, for the region indicated by the user's public geofence. Once in sealed state, the app accesses an SR storing the user's profile and payment details, and receives GUI input from the user about particular connections. It then queries the relevant information from the locally cached database. A similar approach is taken in PlaceLab [28].

If the user wishes to purchase a ticket, the app requests selective declassification of the connection details, along with personal and payment details obtained from the SR. If the user approves, the details are sent to the DB backend as part of a ticketing transaction.

Without the ticketing option, which we have not yet implemented, the app consists of 1417 lines of Java code. Compared to a conventional implementation of the app that has an offline mode, only a few lines of additional code are nec-

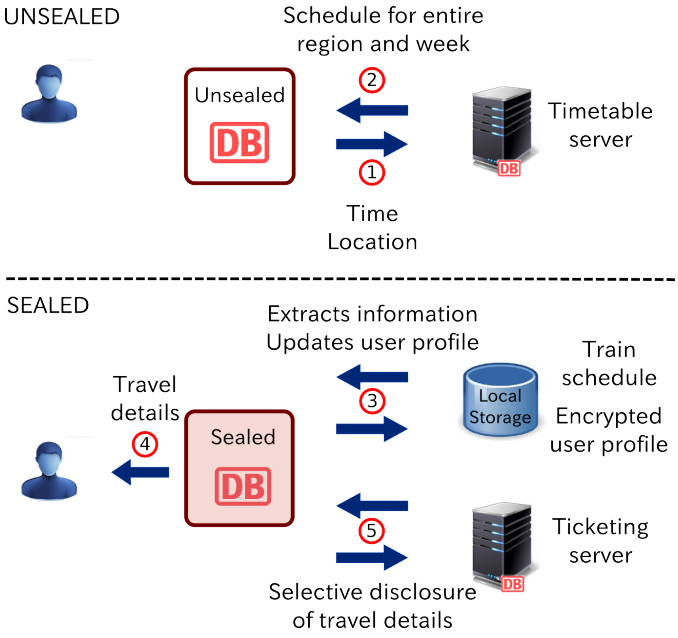

Figure 3: PC-compliant DB app

essary. This code uses the PC API to open an SR, switch to sealed mode, and request selective declassification when issuing a ticketing transaction.

Maintaining a local copy of the database for a region corresponding to the user's current public geofence requires some additional bandwidth. As we will show in Section 5.4, the required bandwidth is low in practice.

4.4 Health/Fitness app

We built a simple wellness/fitness app that monitors a Bluetooth connected wearable fitness sensor (Polar H7 heart rate sensor), and allows users to enter additional information (medications, meals). Users can send some of their data to a coach. We also built a simple app for the coach, who can receive fitness data from users and respond with fitness and nutrition advice. A simple, untrusted web backend service supports a REST API to store encrypted user data and to exchange encrypted data among users and coaches.

The fitness app, in unsealed state, opens an SR used to store the user's profile information and personal health record (PHR), which consists of sensor data and data logged by the user. Then it switches to sealed mode, where it reads out sensors, accepts input from the user, and adds data to the PHR. When the user wishes to send data to a coach, the app requests selective declassification of the corresponding data. If the user approves, the data is encrypted for the coach and sent to the backend service.

The coach's app, in unsealed mode, downloads encrypted user data from the backend. Then, it switches to sealed mode, where it can retrieve its keys from an SR containing the coach's profile data, and decrypts the user data. Once the coach has entered her response, the apps asks for selective declassification of the response.

The apps consist of a library with common procedures (5378 lines of Java code), the client's app (3127 lines of Java code) and the coach's app (314 lines of Java code). There is little code specific to the use of PCs. Usage of the selective declassification API required 202 lines of Java code. Listing 1 shows how to selectively disclose data through a HTTP connection using the PC API.

Listing 1: Example of data declassification through PC HTTP connections

```
...
  PrivacyCapsulesService pcService =
PrivacyCapsulesService.from(getActivityContext());
  long handle = pcService.createKey();
  URL url = new URL("http://example.com/service");
  HttpURLConnection con =
      (HttpURLConnection) url.openDisclosureConnection(handle);
  conn.setRequestMethod("POST");
  ...

  if (pcService.seal()) {
    /* Get account from the contacts database */
    /* Get password from GUI */
    List<NameValuePair> params = new ArrayList<NameValuePair>();
    params.add(new BasicNameValuePair("account", account));
    params.add(new BasicNameValuePair("password", password));

    OutputStream os = conn.getOutputStream();
    BufferedWriter writer =
        new BufferedWriter(new OutputStreamWriter(os,"UTF-8"));
    writer.write(getQuery(params));
    ...
  }
...
```

4.5 Messaging app

We built a simple messaging app that allows the live exchange of text messages among users. In unsealed mode, the app sets up a secure channel, which causes the platform to launch the contacts app to allow the user to select the participants. Once in sealed mode, the app has access to the GUI and sealed channel and can exchange messages with the participants.

The app has around 600 Java lines of code and very few lines are specific to the use of PCs. As shown in Listing 2, the traditional calls to establish a socket (line #3) and to obtain the socket's input and output streams (line #8) are replaced by the PC secure channels API.

Listing 2: Creating a secure channel and obtaining its input/output streams

```
...
  PrivacyCapsulesService pcService =
PrivacyCapsulesService.from(getActivityContext());
  final ParcelFileDescriptor[] contactsFd =
        pcService.createSecureChannel();
  pcService.seal();
  connected = true;
  while (connected) {
    try {
      if (pcService != null) {
        if (contactsFd[0] != null) {
          OutputStream outputStream =
new ParcelFileDescriptor.AutoCloseOutputStream(contactsFd[0]);
          BufferedWriter bWriter =
new BufferedWriter(new OutputStreamWriter(outputStream)),true)
          PrintWriter out = new PrintWriter(bWriter);
          ...
        }
        ...
      }
...
```

5. PERFORMANCE EVALUATION

In this section, we describe a prototype implementation of privacy capsules in Android, and present experiments to evaluate the performance of PCs.

5.1 Prototype

The PC prototype has components in both the Android kernel and the Android framework. In the Android kernel, we added a syscall to support the PC API. Moreover, we added a kernel module (`pc_module.ko`), which contains 3990 lines of code and provides the core of the kernel PC implementation. It intercepts all I/O syscalls of PC-encapsulated processes by changing the syscall table to refer to the PC kernel module [22].

The kernel module also performs the cryptographic operations, and uses AES-128 to encrypt/decrypt the blocks of data. The handle:file mapping is stored by the PC kernel implementation as a system protected file and the key is recovered every time it is required to perform cryptographic operations.

In the Android framework, 3846 lines of code were added to provide a Android system service (PrivacyCapsulesService), notification support for declassification and location policies, permission control, system services control and a Java API for accessing the features provided by PC.

Changes to the existing Android permission system are another part of the prototype. Android has several different permissions that are granted to the application, which are specified in a manifest XML file by the developer. Permissions are managed at two levels: user space permissions and kernel space permissions. For user space, every time the application needs to access a certain resource associated with a specific permission, Android checks if the application has been granted the proper rights to do so. This is implemented in the Android framework. For kernel space, the system associates each permission (for example, GPS access) to a system group. Therefore, only those applications that belong to a particular group (for instance, the gps_access group), can access the resource. The PC implementation makes changes to the permission system to deny apps in unsealed state access to sensitive resources, i.e. calendar, contacts and sensors.

5.2 Experimental platform

The evaluation was performed on a rooted Nexus 9 device (volantis), with Linux Kernel 3.10 (android-tegra-3.10-lollipop-release) and Android 5.1 (AOSP for volantis). The Nexus 9 is a 2.3 GHz dual-core 64-bit Denver tablet, with 2 GB LPDDR3-1600 RAM and 16 GB flash storage.

The Nexus 9 has an ARM V8 CPU with cryptography extension, including SIMD instructions that speed up algorithms such as AES and SHA. To make use of these capabilities, we enabled the following kernel configuration options: CONFIG_CRYPTO_HW, CONFIG_ARM64_CRYPTO, CONFIG_CRYPTO_AES_ARM64_CE, CONFIG_CRYPTO_AES_ARM64_CE_CCM, CONFIG_CRYPTO_AES_ARM64_CE_BLK. For experiments with the Health/Fitness app, we used a Dell Intel®Core™3.20GHz desktop with a i5-3470 CPU and 8 GB of main memory as the backend server. The mobile devices connected to the server via our institute's Wi-Fi network.

5.3 Privacy capsules overheads

We begin with an evaluation of performance overheads of the PC execution model, and the crypto overheads associated with sealed repositories.

In the first experiment, we compare the latency of reading/writing data in different configurations: a plain file in unmodified Android; a plain file with PCs in unsealed state, and a sealed repository in sealed state. Figures 5 and 4 show the results for read and write operations, respectively, of sizes 1, 4K and 10K bytes, with whiskers indicating standard deviation. In each case, an existing file of 1MB size was read or overwritten.

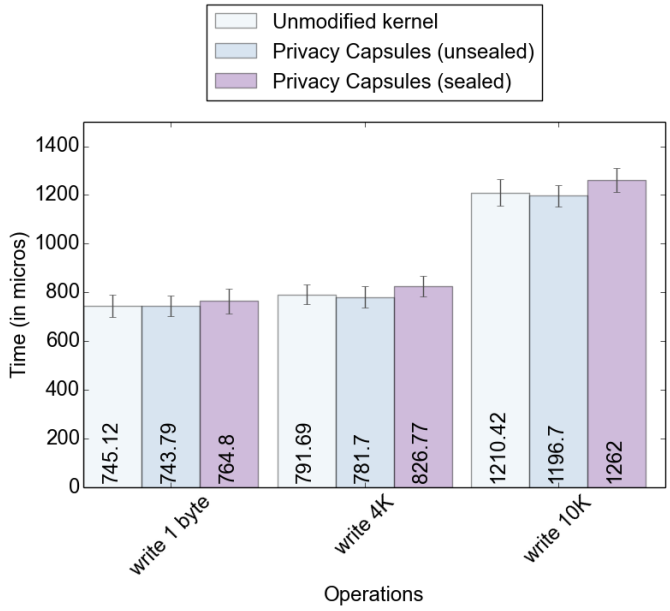

Figure 4: Write overheads

The latency differences between the unmodified Android system and privacy capsules in unsealed state is small, within a standard deviation. This shows that PC's overhead for intercepting I/O operations directly in the kernel is negligible, as expected.

The overheads for read/write operations to SRs are higher, due to the required data encryption. The overhead for write operations is very low, 2.6% for 1 byte, 4.4% for 4K byte, and 4.2% for 10Kbyte writes. Read operations are very fast in our experiments, because they are served from Linux's buffer cache after the first iteration of the experiment. As a result, the overheads are larger, 55% for 1 byte, 319% for 4K byte, and 286% for 10Kbyte reads.

In the second experiment, we compare the performance of two versions of a simple demo app. The two versions are functionally equivalent, but one is PC-compliant, while the other executes on plain Android. The PC-compliant version opens an existing SR and select a key for it, creates a new SR and selects a key for it, then switches to sealed state. In sealed state, it reads the existing SR, writes the new SR, then terminates. The plain Android version opens an existing file and reads it, creates a new file and writes it, then terminates. The SRs and files are 1MB in size.

Figure 6 shows the execution time of the two versions. The values were obtained using traceview [1]. The overhead

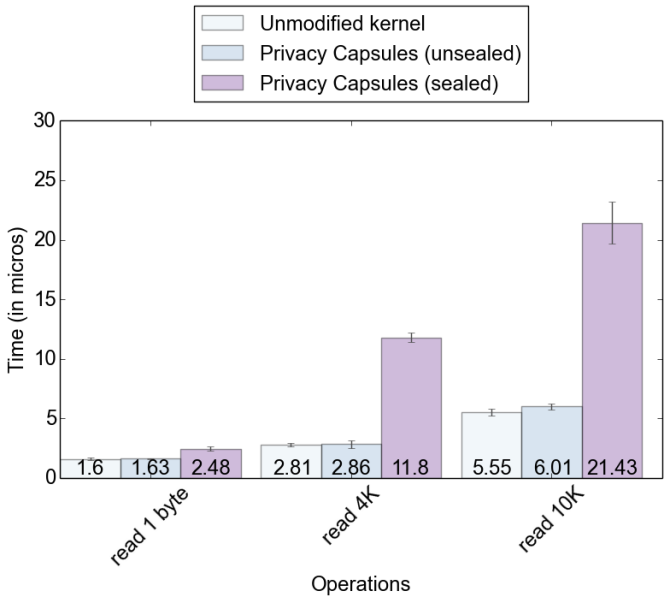

Figure 5: Read overheads

of the PC-compliant version is about 1%.

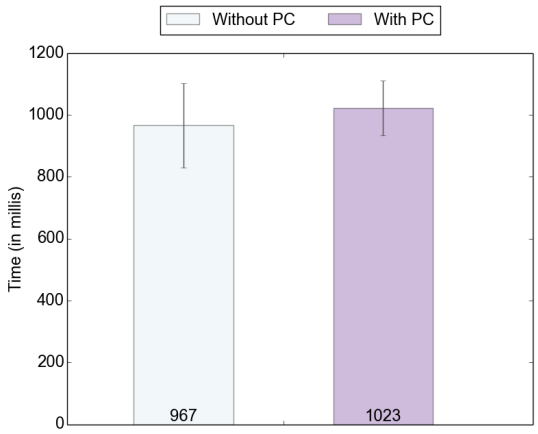

Figure 6: Runtime for two versions of the demo app

We also performed experiments with the sample apps described in Section 4. Latency overheads due to the use of PCs were small enough to remain unnoticeable to the user in all cases.

5.4 Train app: Bandwidth and storage usage

The train timetable app is an example of a class of apps that provide information about a particular geolocation or trajectory from a database. Implementing such apps in a privacy-preserving manner requires that queries about specific locations are performed on a locally cached copy of the database. Minimally, a portion of the database must be cached that is sufficiently large to absorb all queries within the user's public geofence.

The storage and bandwidth requirements for maintaining such a local copy depend on the size and update rate of the database, as well as the size of the user's public geofence. Here, we measure these requirements for the DB train app we built.

Towards this end, we collected data from the Deutsche Bahn (DB) online train schedule system. DB is the largest rail transport company in Central Europe. We fetched the timetable on Dec 5, 2014. It includes 5374 stations, is 12.2MB large, and changes every six months.

The DB systems also publishes real-time updates like platform changes, delays, and cancellations. The updates consist of text messages stating a delay (e.g., "+2min", "appr. +50min", "Train cancelled") and a reason (e.g., "Platform changed", "Technical failure", "Delay from previous trip", "Person on the tracks", "Delay of a preceding train", "Waiting for an incoming train").

We fetched realtime data through the DB Web API for all the stations during the period from 7pm, Mon, 8 December, 2014 to 5:51pm, Wed, 10 December 2014. Of our 144623151 queries, 365843 (or .25%) failed with an HTTP status 404 (not found) or 408 (timeout); as a result, we may have missed a small proportion of realtime updates. The data collected is likely to be typical for weekdays under normal conditions. We expect that volume could be considerably higher in case of a strike, accident, or extreme weather conditions.

We checked how much of the realtime data change over time. Therefore, we considered all changes in the field "delay" and "reason" as updates. Figure 8 shows cumulative distributions of the realtime update volume over time for all German states, and Figure 7 shows the same for the main stations of selected German cities (population in parentheses): Saarbrücken (177,201), Frankfurt am Main (701,350), Berlin (3,562,166) and Hamburg (1,751,775). The figures cover the 24 period within our data collection period that had the highest volume of realtime updates.

We observed many cases of realtime updates that report a delay of zero minutes and a reason of "No delay" for a particular train service. These updates seems redundant, but account for a significant proportion of the updates, ranging from 16.4% for the state of Hamburg to 78% for the state of Sachsen (Saxony).

Given this data, we can estimate the additional bandwidth requirements for a privacy-preserving DB train scheduling app that maintains a cached copy of the timetable and realtime updates relevant to the user's current public geofence. Maintaining the timetable requires up to 12.2 MB twice per year, assuming conservatively that the entire table changes. The resulting bandwidth requirement is negligible.

If a user's public geofence is set to "Germany", maintaining a local cache of realtime updates requires on average 6211 KB for a 24 hour period, which is also negligible. Reducing the public geofence to the state level reduces the bandwidth requirement to between 19 KB/24 hours for Bremen (state with least volume) and 1138 KB/24 hours for North Rhine-Westphalia (state with most volume). Reducing the public geofences further to the city level reduces the bandwidth requirement to between 7.92 KB/24 hours for Saarbrücken (city with lowest volume) and 60.34 KB/24 hours for Frankfurt am Main (city with highest volume).

We conclude that the bandwidth requirements for maintaining a local cache of timetable and realtime updates for a train schedule app is neglibigle, even when the user's public geofence is at the level of a large country in Europe. Given these very low traffic volumes, even unusual events (strike,

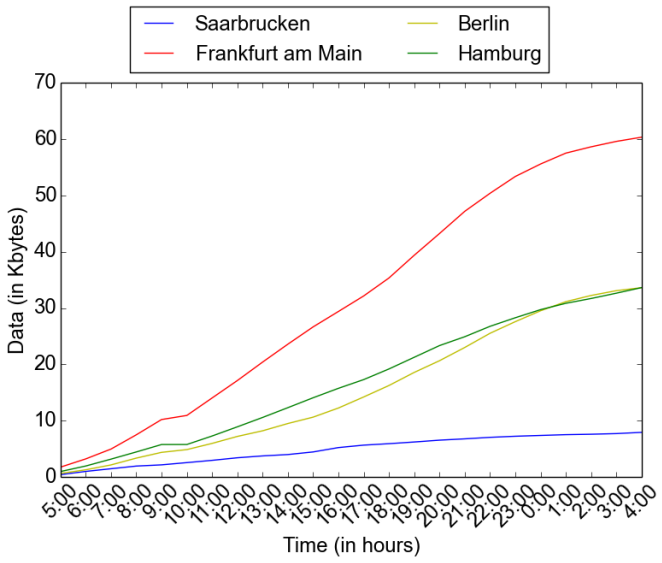

Figure 7: Cumulative DB realtime data by city

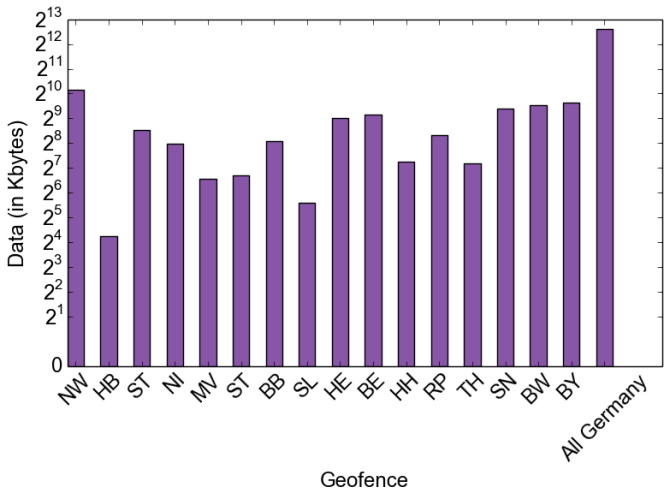

Figure 8: DB realtime data over 24 hours, by state

weather) that cause the volume to increase by one or two orders of magnitude would be of little concern. Note that the use of a public geofence larger than country would make no difference in this case, as the scope of the DB system and app is limited to a country.

6. CONCLUSIONS

We presented the design of privacy capsules, an execution platform for mobile apps that prevents privacy leaks by design. We have implemented a prototype implementation in Android and experimented with three example applications, representing common application classes. Runtime overhead is very low, and the app changes required to program apps within the privacy capsules execution model are moderate.

7. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers and our shepherd Landon Cox for their helpful feedback. This research was partially supported by the European Research Council (ERC Synergy imPACT 610150), the German Science Foundation (DFG CRC 1223), and by the United States National Science Foundation (award numbers: IIS-0964541 and CNS-1314857.)

8. DEDICATION

We dedicate this paper to the memory of Scott DellaTorre.

9. REFERENCES

[1] Android traceview tool. http://developer.android.com/tools/help/traceview.html. Accessed: 08.12.2015.

[2] Meet Bump, the App Store's Billionth Download. http://www.pcworld.com/article/163840/article.html. Accessed: Apr 11, 2016.

[3] MetaIntell identifies enterprise security risks, privacy risks and data leakage in 92% of top 500 android mobile applications. http://www.businesswire.com/news/home/20140122006295/en/MetaIntell-Identifies-Enterprise-Security-Risks-Privacy-Risks#.VLeLfnWx15Q. Accessed: Jan 15, 2015.

[4] Mobile malware stealing data from legitimate apps. http://www.alphr.com/news/security/389716/mobile-malware-stealing-data-from-legi timate-apps. Accessed: Nov 22, 2015.

[5] Review: Bump, the One Billionth iPhone App. http://www.pcmag.com/article2/0,2817,2345899,00.asp. Accessed: Apr 11, 2016.

[6] The Top Ten Mobile Flashlight Applications Are Spying On You. Did You Know? http://www.cyberdefensemagazine.com/the-top-ten-mobile-flashlight-applications-a re-spying-on-you-did-you-know/. Accessed: Aug 31, 2015.

[7] UNH cyber forensics group reveals smartphone app issues affecting 968 million. http://www.unhcfreg.com/#!UNH-Cyber-Forensics-Group-Reveals-Smartphone-App-Issues-Affecti n g-968-Million/c5rt/376C3F2A-18F6-41E9-9A42-D05AAD8E2DCA. Accessed: Jan 15, 2015.

[8] Aditya, P., Sen, R., Joon Oh, S., Benenson, R., Bhattacharjee, B., Druschel, P., Wu, T., Fritz, M., and Schiele, B. I-Pic: A Platform for Privacy-Compliant Image Capture. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2016), MobiSys '16, ACM.

[9] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. *SIGPLAN Not. 49*, 6 (June 2014), 259–269.

[10] Atzeni, A., Su, T., Baltatu, M., D'Alessandro, R., and Pessiva, G. How dangerous is your android app?: An evaluation methodology. In *Proceedings of the 11th International Conference on Mobile and*

Ubiquitous Systems: Computing, Networking and Services (ICST, Brussels, Belgium, Belgium, 2014), MOBIQUITOUS '14, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), pp. 130–139.

[11] Au, K. W. Y., Zhou, Y. F., Huang, Z., and Lie, D. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 217–228.

[12] Backes, M., Gerling, S., Hammer, C., von Styp-Rekowsky, P., and Maffei, M. Appguard - real-time policy enforcement for third-party applications. Tech. Rep. A/02/2012, Saarland University, http://www.infsec.cs.uni-saarland.de/projects/appguard/android_irm.pdf, July 2012.

[13] Baden, R., Bender, A., Spring, N., Bhattacharjee, B., and Starin, D. Persona: An online social network with user-defined privacy. *SIGCOMM Comput. Commun. Rev. 39*, 4 (Aug. 2009), 135–146.

[14] Bartel, A., Klein, J., Le Traon, Y., and Monperrus, M. Automatically securing permission-based software by reducing the attack surface: An application to android. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2012), ASE 2012, ACM, pp. 274–277.

[15] Bichhawat, A., Rajani, V., Garg, D., and Hammer, C. Information flow control in webkit's javascript bytecode. In *Principles of Security and Trust*, M. Abadi and S. Kremer, Eds., vol. 8414 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2014, pp. 159–178.

[16] Cheng, W., Ports, D. R. K., Schultz, D., Popic, V., Blankstein, A., Cowling, J., Curtis, D., Shrira, L., and Liskov, B. Abstractions for Usable Information Flow Control in Aeolus. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Berkeley, CA, USA, 2012), USENIX ATC'12, USENIX Association, pp. 12–12.

[17] Cox, L. P., Gilbert, P., Lawler, G., Pistol, V., Razeen, A., Wu, B., and Cheemalapati, S. Spandex: Secure password tracking for android. In *Proceedings of the 23rd USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2014), SEC'14, USENIX Association, pp. 481–494.

[18] Efstathopoulos, P., Krohn, M., VanDeBogart, S., Frey, C., Ziegler, D., Kohler, E., Mazières, D., Kaashoek, F., and Morris, R. Labels and Event Processes in the Asbestos Operating System. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2005), SOSP '05, ACM, pp. 17–30.

[19] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–6.

[20] Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2011), CCS '11, ACM, pp. 627–638.

[21] Felt, A. P., Ha, E., Egelman, S., Haney, A., Chin, E., and Wagner, D. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security* (New York, NY, USA, 2012), SOUPS '12, ACM, pp. 3:1–3:14.

[22] Garfinkel, T. Traps and pitfalls: Practical problems in system call interposition based security tools. In *In Proc. Network and Distributed Systems Security Symposium* (2003), pp. 163–176.

[23] Gerber, P., Volkamer, M., and Renaud, K. Usability versus privacy instead of usable privacy: Google's balancing act between usability and privacy. *SIGCAS Comput. Soc. 45*, 1 (Feb. 2015), 16–21.

[24] Gibler, C., Crussell, J., Erickson, J., and Chen, H. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing* (Berlin, Heidelberg, 2012), TRUST'12, Springer-Verlag, pp. 291–307.

[25] Henderson, A., Prakash, A., Yan, L. K., Hu, X., Wang, X., Zhou, R., and Yin, H. Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2014), ISSTA 2014, ACM, pp. 248–258.

[26] Jeon, J., Micinski, K. K., Vaughan, J. A., Fogel, A., Reddy, N., Foster, J. S., and Millstein, T. Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices* (New York, NY, USA, 2012), SPSM '12, ACM, pp. 3–14.

[27] King, D., Hicks, B., Hicks, M., and Jaeger, T. Implicit flows: Can't live with 'em, can't live without 'em. In *Proceedings of the 4th International Conference on Information Systems Security* (Berlin, Heidelberg, 2008), ICISS '08, Springer-Verlag, pp. 56–70.

[28] LaMarca, A., Chawathe, Y., Consolvo, S., Hightower, J., Smith, I., Scott, J., Sohn, T., Howard, J., Hughes, J., Potter, F., Tabert, J., Powledge, P., Borriello, G., and Schilit, B. Place lab: Device positioning using radio beacons in the wild. In *Proceedings of the Third International Conference on Pervasive Computing* (Berlin, Heidelberg, 2005), PERVASIVE'05, Springer-Verlag, pp. 116–133.

[29] Lee, S., Wong, E. L., Goel, D., Dahlin, M., and Shmatikov, V. πBox: A platform for privacy-preserving apps. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2013),

nsdi'13, USENIX Association, pp. 501–514.

[30] Lu, L., Li, Z., Wu, Z., Lee, W., and Jiang, G. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 229–240.

[31] Ren, J., Rao, A., Lindorfer, M., Legout, A., and Choffnes, D. ReCon: Revealing and Controlling PII Leaks in Mobile Network Traffic. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2016), MobiSys '16, ACM.

[32] Schwartz, E. J., Avgerinos, T., and Brumley, D. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2010), SP '10, IEEE Computer Society, pp. 317–331.

[33] Singh, K., Bhola, S., and Lee, W. xbook: Redesigning privacy control in social networking platforms. In *Proceedings of the 18th Conference on USENIX Security Symposium* (Berkeley, CA, USA, 2009), SSYM'09, USENIX Association, pp. 249–266.

[34] Sun, M., and Tan, G. NativeGuard: Protecting Android Applications from Third-party Native Libraries. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks* (New York, NY, USA, 2014), WiSec '14, ACM, pp. 165–176.

[35] Viswanath, B., Kiciman, E., and Saroiu, S. Keeping information safe from social networking apps. In *Proceedings of the 2012 ACM Workshop on Workshop on Online Social Networks* (New York, NY, USA, 2012), WOSN '12, ACM, pp. 49–54.

[36] Xu, R., Saïdi, H., and Anderson, R. Aurasium: Practical Policy Enforcement for Android Applications. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2012), Security'12, USENIX Association, pp. 27–27.

[37] Zeldovich, N., Boyd-Wickizer, S., Kohler, E., and Mazières, D. Making Information Flow Explicit in HiStar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 19–19.