

Copyright  
by  
Deepak Goel  
2013

The Thesis Committee for Deepak Goel

Certifies that this is the approved version of the following thesis:

**A CleanRoom Approach To Bring Your Own Apps**

APPROVED BY

SUPERVISING COMMITTEE:

---

Michael Dahlin, Supervisor

---

Vitaly Shmatikov

**A CleanRoom Approach To Bring Your Own Apps**

by

**Deepak Goel, B.Tech.**

**THESIS**

Presented to the Faculty of the Graduate School of  
The University of Texas at Austin  
in Partial Fulfillment  
of the Requirements  
for the Degree of

**MASTER OF SCIENCE IN COMPUTER SCIENCE**

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2013

Dedicated to my parents, teachers, and wife  
For their endless love, support, and encouragement

## Acknowledgments

First and foremost, I would like to express my sincere gratitude to my advisor, Professor Mike Dahlin, for giving me an opportunity to work on this project and mentoring me throughout. I also want to thank Professor Vitaly Shmatikov, for providing feedback at various stages of the project and reviewing this document.

This thesis would not have been possible without the help from my teammates Sangmin Lee and Edmund Wong, with whom I had numerous technical discussions. I share the credit of my work with Sangmin Lee for his significant contributions to the design and implementation of CleanRoom. He designed and implemented the sandboxes both on the device by modifying the Android kernel, and on the cloud by using jetty webserver, created wrappers around HBase for providing app and blob storage, and evaluated the performance of CleanRoom. <sup>1</sup>

I am thankful to William Young for supporting me through graduate research assistantship.

Finally, I would like to thank my wife Ankita, who has been hugely supportive throughout my graduate studies.

---

<sup>1</sup>Please refer to Appendix for more details on individual contributions.

# **A CleanRoom Approach To Bring Your Own Apps**

Deepak Goel, M.S.Comp.Sci.  
The University of Texas at Austin, 2013

Supervisor: Michael Dahlin

Today, on mobile devices such as smartphones and tablets, hundreds of thousands of software apps provide useful services to users. Users use these apps to search and browse the web, perform financial transactions, emailing, among other functions. Besides, these apps use cloud services which gives the users the flexibility to access them from anywhere and from any device. Because of the rich functionality of these apps and ease of use of mobile devices, users (employees) often want to use their devices and preferred apps at their workplace. However, these apps not only pose risk to user's private data but also to enterprise data, when users use them within an enterprise network. For one thing, these apps come from hundreds and thousands of different app publishers, where all of them may not be trustworthy. Second, apps often need user's private data such as location, contact list, photos among others and use remote cloud to carry out their operations. In the process apps may leak a user's private or enterprise confidential data to a third party.

Current practices to prevent such leaks through user enabled app permissions fall short because often user does not understand these permissions.

Besides, even if a company's "Bring Your Own Device" (BYOD) policies mitigate the risk of device compromise with enterprise-approved password policies, remote wipe capabilities, and OS security upgrade policies, the apps on those devices pose their own risks.

This thesis presents CleanRoom, a new app platform that prevents apps from leaking the data entrusted to them. It does not rely on users to make good decisions about Privacy, and enables enterprises to allow its employees to use their own devices and bring their preferred apps to work.

# Table of Contents

<b>Acknowledgments</b>	<b>v</b>
<b>Abstract</b>	<b>vi</b>
<b>List of Tables</b>	<b>x</b>
<b>List of Figures</b>	<b>xi</b>
<b>Chapter 1. Introduction</b>	<b>1</b>
<b>Chapter 2. Background</b>	<b>9</b>
2.1 Privacy . . . . .	9
2.2 Information Flow . . . . .	12
<b>Chapter 3. Threat Model</b>	<b>14</b>
<b>Chapter 4. Design</b>	<b>18</b>
4.1 App components and sandboxes . . . . .	18
4.2 Storage and communication . . . . .	20
4.3 Privacy preserving communications . . . . .	21
<b>Chapter 5. Differential Privacy</b>	<b>27</b>
<b>Chapter 6. Information Flow Model</b>	<b>33</b>
6.1 Secrecy . . . . .	37
6.2 Case Study: Text Editor app . . . . .	39
6.3 Characteristics of the model . . . . .	40

<b>Chapter 7. Implementation</b>	<b>42</b>
7.1 Background . . . . .	42
7.1.1 Android . . . . .	42
7.1.2 Jetty WebServer . . . . .	45
7.1.3 Hbase . . . . .	46
7.2 CleanRoom Implementation . . . . .	46
7.2.1 Isolations and authentication . . . . .	47
7.2.2 Storage and communication channels . . . . .	51
<b>Chapter 8. Applications</b>	<b>52</b>
8.1 Text Editor . . . . .	52
8.2 Calendar . . . . .	57
8.3 Instant Messaging . . . . .	60
<b>Chapter 9. Design Guidelines</b>	<b>62</b>
<b>Chapter 10. Evaluation</b>	<b>65</b>
10.1 Performance overhead . . . . .	65
10.2 Privacy vs accuracy . . . . .	67
<b>Chapter 11. Related Work</b>	<b>69</b>
<b>Chapter 12. Future Work</b>	<b>74</b>
<b>Chapter 13. Conclusion</b>	<b>77</b>
<b>Appendix</b>	<b>78</b>
<b>Appendix 1. Contributions</b>	<b>79</b>
<b>Index</b>	<b>81</b>
<b>Bibliography</b>	<b>82</b>

# List of Tables

5.1	Parameters for counters. . . . .	32
-----	----------------------------------	----

## List of Figures

1.1	Android Permissions. . . . .	3
3.1	CleanRoom threat model . . . . .	15
4.1	Design overview. . . . .	19
4.2	CleanRoom design. . . . .	22
4.3	Sharing in CleanRoom . . . . .	24
5.1	Algorithm for delayed-output counter. . . . .	30
6.1	Privilege hierarchy . . . . .	34
8.1	Text Editor Design . . . . .	54
8.2	Text Editor application screenshots . . . . .	56
8.3	Calendar application screenshots . . . . .	59
8.4	Instant Messaging application screenshots . . . . .	61
10.1	Latency vs. Throughput for CleanRoom mechanisms. . . . .	66
10.2	Overhead of isolation for various workloads. . . . .	67
10.3	Accuracy of delayed-output counter . . . . .	68

# Chapter 1

## Introduction

This thesis describes a new way to reduce the risk to privacy of users and enterprises when using applications provided by untrusted publishers to manipulate private data. In the proposed architecture, the platform enforces privacy; it does not rely on users to make good decisions about privacy, and it does not assume that app developers are all trustworthy.

Today, on mobile devices such as smartphones and tablets, hundreds of thousands of software apps provide useful services to users. Users use these apps to perform searches, navigations, financial transactions, emailing, among other functions. Besides, these apps use cloud services which gives the users the flexibility to access these apps from anywhere or from any device. Because of the rich functionality of these apps and ease of use of mobile devices, users (employees) often want to use their devices and preferred apps at their workplace. To cater to these users, enterprises have been increasingly adopting “Bring Your Own Device” (BYOD) and “Bring Your Own Apps” (BYOA) policies, which allow employees to bring their personal devices (BYOD) and use their preferred apps (BYOA) at work. These policies enable greater employee engagement and satisfaction, as well as improves productivity.

However, these apps not only pose risk to user's private data but also to enterprise data when users use them within an enterprise network. For one thing, these apps come from hundreds and thousands of different app publishers, where all of them may not be trustworthy. Moreover, these apps access user's private data (e.g. contacts, location, photos), sensor inputs (e.g. camera, microphone, GPS), or information about user's behavior (e.g. tracking user's likes and dislikes) and use remote cloud to carry out their operations. While most apps use this data responsibly, but there have been a number of incidences where apps violated the privacy of the users and enterprise by exfiltrating the data to third party [8, 9, 15] [17, 43].

#### **Limitations of current practices.**

- *App permission* - Most current app platforms such as iOS and Android rely on users to explicitly grant permissions to apps as shown in Figure 1.1. This approach is deeply flawed. Users are not in a position to decide which apps to trust with what resource. At best, they must consider scores of apps, contemplate their purpose, and then decide whether to trust them with particular resource. In reality, the situation is even worse because an app may request a permission that is necessary for the app (e.g. a map application that accesses a user's location) and then use that permission in an unexpected way (e.g. sell history of user's movement over the year).

Moreover, users - who are inundated with permission requests and may not fully understand the implications as to how an app can misuse the given permission - often blindly grant all requests [32] or even disable notifications [44] implicitly entrusting all apps with their private data.

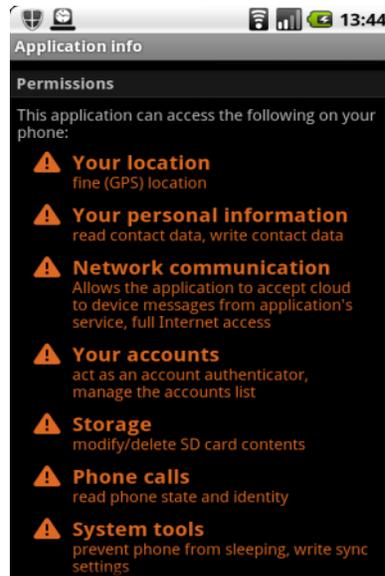


Figure 1.1: Android Permissions.

- *Blacklist & Whitelist apps* - Enterprise IT departments often restrict the apps the employees can use through *blacklists* - allow all apps except those in the blacklist, (e.g., [7]) and *whitelists* - deny all apps except those in the whitelist, (e.g., [5]). Apps are often complex and frequently updated, which makes these approaches:
  - Expensive - an app review process is likely to be feasible only for organizations with a sizeable IT department and for a modest number

of apps;

- Ineffective - the app review process is tedious and prone to errors [3, 9]; and
- Counter productive - any review process will likely to be conservative and will prevent adoption of many useful and safe apps [2, 7].

- *Partition of workspaces* - In this approach the device is partitioned into personal and business workspaces through the use of virtual machines [16, 20]. However, partitioning does not prevent apps from leaking an enterprise's confidential information that they have access to or accessing information that they should not, so the business workspace must still be secured as before.

This thesis presents CleanRoom, a new app platform with the primary aim to protect enterprise confidentiality from careless or malicious app publishers. CleanRoom enforces two main enterprise privacy policies:

1. An app cannot leak an enterprise's data such as emails, financial documents, project related documents, to outsiders, including the app publisher.
2. An app cannot leak an enterprise's data to an employee who does not have access to it.

The above two policies signify that an app cannot increase the set of users with access to a given data. Only the owner (user or enterprise) of the

data can do so. CleanRoom allows the owner to declassify the data if the owner wants to do so but in that case it would be the owner's decision and not of the app.

CleanRoom achieves its goals by shifting the trust from many apps and their publishers to few well-known platform providers running CleanRoom and by enforcing isolation at two levels:

1. It isolates each user's instance of an app from other instances, and
2. It isolates each document within an app from other documents.

Moreover, communication between these isolated units is through few well-defined storage and communication channels. Because these channels are controlled by CleanRoom, CleanRoom can give rigorous privacy guarantees about the information that flows through them.

CleanRoom is not a privacy panacea but it shifts the trust to a better place. It seems reasonable to ask users and enterprises to trust the platform for at least three reasons. First, the platform provider may often be associated with a trusted brand (e.g. Google, Apple, Amazon, or Facebook). Second, it is safer to trust few well-known brands than to trust hundreds of thousands of app publishers. Third, even without CleanRoom the user has to trust the platform provider (e.g. user has to trust her phone to keep her personal data).

Moreover, CleanRoom trusts a user that has access to the data, to use it responsibly. If she does not, then it is possible to leak the data even under CleanRoom.

CleanRoom uses four novel techniques to achieve its goals:

1. A per-user user sandbox that spans a user’s device and a cloud back-end. The latter may be supplied by the device’s platform provider (e.g. Google or Apple) or the user’s employer.
2. A per-blob data sandbox that spans multiple users’ devices and a cloud back-end.
3. Four specialized storage and communications systems that enable a variety of apps to do useful work within CleanRoom while preserving user privacy.
4. An adaptation and implementation of *differential privacy under continual observation* that improves the trade-off between accuracy and privacy of released statistics.

CleanRoom’s sandboxes span the device and the cloud which enables enterprises to deploy BYOA policies and allow its employees to execute apps from untrusted publishers on a trusted platform. This platform may run on the premises under the enterprise’s direct control or be part of an external ”app store” or hosting infrastructure. Similar to BYOD policies, where companies install profiles and security software on employee-owned devices used for work, a company might restrict apps to run only within CleanRoom, thus ensuring that these apps and any information they access, are securely confined.

The deployment of CleanRoom benefits *everyone*. Employees using CleanRoom on their devices have the freedom to use the apps they prefer. By allowing employees to use their preferred apps, enterprises can maximize employees' efficiency while not having to worry about the privacy of its data. Even app publishers and the platform provider (e.g., Google or Apple) benefit as CleanRoom enables many more users to use more apps on their devices.

However, CleanRoom has some downsides. It requires apps to be written under the new platform which also mean that existing apps have to be rewritten. Users would face a period of transition where all the apps may not be available under the new platform. Also, it may be possible that certain app features may be difficult or impossible to implement under CleanRoom and certain user interface features may become complicated.

This thesis provides an overview of CleanRoom and how it enables collaboration among employees while preserving enterprise privacy. It then presents an information flow control (IFC) model of CleanRoom to show that:

1. CleanRoom design resulted in a simplified IFC model
2. CleanRoom restricts an app from exfiltrating an enterprise confidential information both within and outside the organization.

To demonstrate that useful apps can be developed on CleanRoom, this thesis describes three prototype apps that were developed on CleanRoom and provides a set of design guidelines for developing more apps. The performance

evaluation of CleanRoom demonstrates that CleanRoom incurs a very modest overhead and can be used in a variety of practical scenarios.

# Chapter 2

## Background

*This chapter briefly describes privacy and information flow in a general sense, which would be useful later when these topics are discussed in the context of CleanRoom*

### 2.1 Privacy

Information security refers to the act of preventing unauthorized access, use, or disclosure of the data, disruption or unintended modification to the data, and assuring that the data is from purported owner.

Preventing unauthorized access or disclosure of information is referred to as *privacy*. It assures that the information is only available to its rightful users. For example, in a multiuser operating system such as Linux, where it is possible that many users can simultaneously use the system, a user has to first authenticate herself by providing a username and password to establish her identity, only then she is allowed to use the system. Moreover, Linux makes sure that each authenticated user is able to access only the data that she is authorized to access.

Methods used for data privacy can be classified into two broad cate-

gories:

- *Discretionary Access Control* (DAC) refers to a type of access control in which access to an object (e.g. a file) is based on the identity of the subject (e.g. username). It is 'discretionary' in the sense that the subject under certain conditions (e.g. if the subject is the creator of the object) can share/transfer the access control to other subjects. An example of DAC is seen in operating systems such as Linux, where each file is associated with access permissions for user, group and others. A user can control the access to a file by changing access permission either for the group or others.
- *Mandatory Access Control* (MAC) refers to an access control where there is an authority which decides which subject should access which object. Unlike DAC, subject in MAC cannot decide on their own to share /transfer their access control to other subjects. MAC implementation can be found in SELinux [14]

*Integrity* refers to accuracy and validity of the data. In a lifetime of a data, the data may be stored, updated, retrieved or transferred, *integrity* ensures that data remains identical to what was inputted initially or at the time of update, and it remain unaltered in transit from source to destination during retrieval or transfer. There are a number of methods usually employed to provide data integrity, such as:

- *Input validation* - This method checks the correctness of input values before storing or modifying the original data. For example, in a database system if some attribute can only have numeric values, then the input value should not contain non-numeric values.
- *Data validation* - This method ensures that the data received from a remote source is identical to the data sent by the source. It prevents alteration to data in transmit.
- *Data encryption* - This method uses cryptography to lock the data with a cipher and prevents undetected modification to the data.

*Authenticity* refers to the act of ensuring that the data has been created by the claimed owner. For instance, whenever there is a software update from Adobe, the system first authenticates that the update is actually from Adobe and not from some impostor.

The main focus of CleanRoom is *privacy*. In particular, in CleanRoom the right to access an enterprise's confidential information is given by the enterprise to its employees. CleanRoom enforces a combination of mandatory access control (MAC) and discretionary access control (DAC) policies in which an employee is only allowed to access the data to which enterprise has given her the permission (MAC). However, once an employee has an access to certain data, the enterprise trusts employee's discretion (DAC) to share the access of the data with other employees or outside of the organization. Thus, CleanRoom restricts access to information to an authorized employee but does not

control what an employee does with that information. Chapter 3 describes the privacy model of CleanRoom in more details.

## 2.2 Information Flow

Information flow is the transfer of information from one object to another object. For instance, in an operating system, when a process reads a file, information flows from that file to the process. An Information flow control (IFC) model defines what *information* flows and in which *direction*.

An IFC model provides stronger security guarantees than that provided by access control list, firewall or cryptography. These methods control the access to information but once the access is made, they have no control on how that information is used or propagated. For example, a user who has access to a file can copy the content of that file into some other file.

Various IFC models have been developed that enforce information flow rules at different abstractions. On one hand, there are IFC models that are applied at language level [25]. These models usually use static analyzer on annotated code to ensure that information flow within a program is according to the IFC policies. On the other hand, there are IFC models that are applied at process level [30, 39, 55]. Unlike IFC models at language level, IFC at process level is dynamic and uses tags and labels to track the flow of information within the system.

Besides granularity (process or language level), an IFC model can be

centralized or decentralized. Like MAC, in centralized IFC model [25], there is a central authority (e.g. government or military) that defines rules for information flow within the system. Typically, in such models information is classified into various security classes and information flow rules are defined among these classes. For instance, a military organization may classify information as *unclassified*, *confidential*, *secret*, and *top secret* and define information flow rule such that information can flow from unclassified to confidential to secret to top secret but not in other direction.

On the other hand, there are decentralized IFC models [30, 39, 55] in which each participating process defines its own information flow rules to protect its own data. It is 'decentralized' in the sense that there is no central authority that defines the information flow rules for the entire system.

CleanRoom IFC model is a dynamic model which is applied at an abstraction of sandboxes. It is a hybrid model as there is a central authority (enterprise) that defines the information flow rules for all of its data. Within these rules, each employee can then define her own rules to limit the flow of data owned by her. Chapter 6 explains CleanRoom IFC model in more details.

## Chapter 3

# Threat Model

*This chapter describes the principals involved in CleanRoom and the trust that CleanRoom assumes for each of these principals. Based on the trust model, this chapter then discusses the possible threats and the protections that CleanRoom provides.*

As shown in Figure 3.1, there are four principals in CleanRoom:

- *Platform provider* provides the underlying platform for apps running on user's local device and/or the cloud; common examples include Google (Android and Cloud platform) and Apple (iOS and iCloud).
- *App publishers* develop the apps that users install on their devices.
- *Users* are employees of a particular enterprise and may use their devices to access enterprise data via installed apps.
- *Enterprise* decides privacy policies which are enforced by platform.

The goal of CleanRoom is to protect the enterprise's data. In this model, the platform provider is trusted, employees are trusted with the data they have access to, but the app publishers may be malicious. Apps may

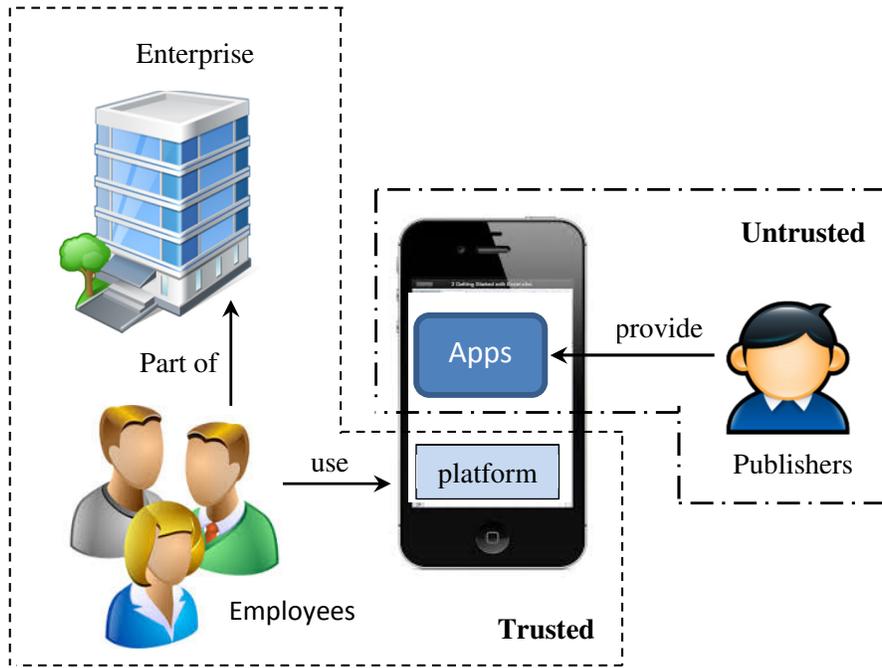


Figure 3.1: CleanRoom threat model

use cloud resources, but these cloud resources are assumed to be running in a private cloud deployment hosted by the enterprise or the platform provider.

The code of the app may attempt to leak enterprise’s private data to an employee who does not have access to that data or reveal information outside the enterprise. That said, the attacker is subject to standard computational feasibility constraints (e.g., the attacker cannot subvert cryptographic primitives). Employees may collude with the app publishers in an attempt to learn data which they do not have access to, e.g., employee trying to learn other employee’s salary.

In CleanRoom, apps operate on *blobs* which may, in reality, consist of multiple pieces of content and data, (e.g., a calendar is a blob that consists of multiple appointments, tasks, etc.), to which users are granted read-only or read-write access. Each blob, has a set of authorized users—the blob creator and the set of users explicitly added by the creator or administrators—whom the enterprise trusts with access to the blob. CleanRoom provides privacy guarantees to each blob. It ensures that the blob is accessible only to users who are in the set of authorized users. However, CleanRoom does not control what these authorized users do with the blob. For instance, an authorized user can copy the data from one blob to another blob or can email it to her friends outside of the organization.

CleanRoom provides extended sandboxes that span from the device to the cloud. Like any software, if CleanRoom is implemented incorrectly, it may be subject to code injection and other attacks that compromise the “ideal sandbox” abstraction. These attacks are outside the scope of this work, which focuses primarily on the design of the sandboxes. Another way in which the “ideal sandbox” abstraction may be violated is via covert (e.g., timing) channels between processes running in the sandbox and those outside the sandbox [40, 49]. If an implementation of CleanRoom is vulnerable to such channels, apps may be able to exfiltrate private data.

CleanRoom uses differential privacy to report aggregate statistics of an app to its publisher (e.g. ad impression counts). This channel leaks a little information with every output, which is inevitable with differential privacy.

However, no covert communication is possible beyond this leakage because by definition, differential privacy holds regardless of the auxiliary information available to the recipient. Chapter 5 discusses differential privacy as used by CleanRoom in detail and quantifies leakage through this channel.

# Chapter 4

## Design

*This chapter presents a detailed design of CleanRoom. It also discusses how CleanRoom enables collaboration among the employees of an organization, while preserving privacy of the enterprise data*

### 4.1 App components and sandboxes

In CleanRoom, an app consists of two components - *viewer* and *editor*. Viewer is the app component that deals with creating or reading the data, whereas in the editor component, an app can modify the data. For example, in a calendar app, viewer can be used to create a new calendar or read events from existing calendar, and an editor can be used to update the calendar. At the time of development, an app publisher divides the app into multiple components and provides a manifest file that specifies the type of each component based on its functionality.

As shown in Figure 4.1, each of these components may have two halves: one runs locally on the user's device, and the other runs remotely in the cloud. The local half of an app component running on the user's device can only connect to the remote half associated with the same app component.

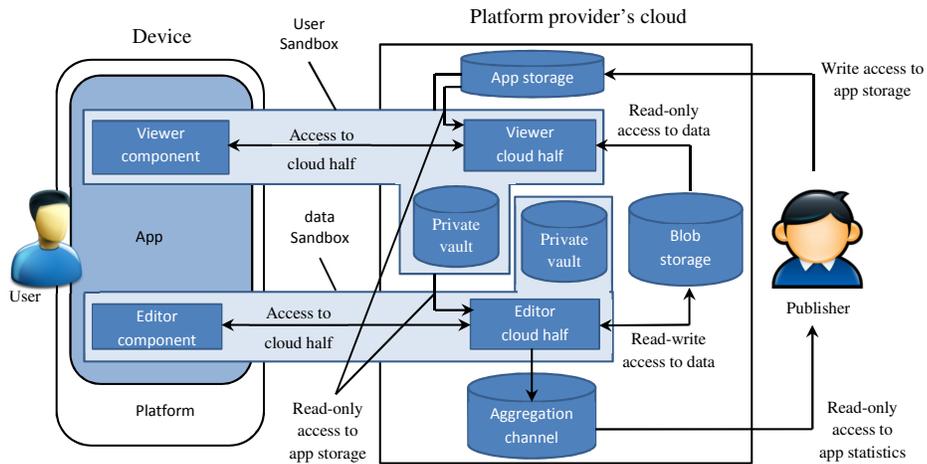


Figure 4.1: Design overview.

CleanRoom, as a platform, runs both on the device and in the cloud and supplies two sandboxes - a per-user *user sandbox* which has read-only access to the data and is used to run the viewer component of an app, and a per-blob *data sandbox*, which has read-write access to the data and runs editor component of the app. Multiple viewer components corresponding to different apps and belonging to the same user could be running in a user sandbox. Similarly, multiple editor components of different apps can run in a single data sandbox. Moreover, these sandboxes span from user device to the cloud and provide an abstraction that a slice of the cloud is part of the user's device: all of the app's computations and storage are done within this "distributed" device, which is otherwise isolated to protect the user's privacy.

## 4.2 Storage and communication

App components running within CleanRoom cannot write data or establish network connections outside of their sandboxes. However, in order to support useful app functionality, CleanRoom provides four restricted storage and communication channels (see Figure 4.1).

The *private vault* provides per-sandbox storage that lets an app component store data specific to a particular user (e.g. user profile, location, query, history etc.), in order to provide personalized services. For example, a calendar app can store information about customized look and feel of a calendar, based on user settings. Each sandboxed app component has read-write access to its own private vault; no other component has any access to it.

The *blob storage* provides a per-blob storage. In CleanRoom a blob is the smallest unit of data for which sharing is supported. It is an abstraction which in reality could be multiple pieces of data. For example, in a calendar app, a user calendar is a blob, in an instant messaging app, conversation with each contact is a blob. At the time of app development the app developer defines blob for the app. User sandbox has read-only and data sandbox has read-write access to blob storage. User sandbox can read from multiple blob storage but cannot write to it.

The *content storage* provides per-publisher storage for the content that the app instances need to function e.g., maps for a navigation app. Each publisher has read-write access to its own content storage so that the publisher

can update the content. CleanRoom grants read-only access of content storage to apps that need the content. Although content storage is shared across all the sandboxes that have access to it, read-only access prevents communication between these sandboxes.

The *aggregation channel* provides a per-app channel (shared among all instances of an app) for publishers to collect statistics on users' collective behavior while protecting privacy of individual blobs. For example, publishers of news apps may learn which articles are popular, but not who viewed what content. Publishers have read access to their respective aggregate channels. However, only app editor component running in data sandbox has write access to aggregate channel. Moreover, aggregate channel uses differential privacy under continual observation [29] to protect the privacy of individual blob. Use of Differential privacy in CleanRoom is discussed in details in Chapter 5

### 4.3 Privacy preserving communications

CleanRoom aims to enable as much functionality as possible while safeguarding the privacy of the users or the enterprise. CleanRoom achieves this goal through the various sandboxes and restricted communication between app components and storage channels. Figure 4.2 shows all possible communications that can happen in CleanRoom.

When a user starts an app in CleanRoom on a local device, the app's local half first authenticates itself (flow *a* in the Figure 4.2) by making a request to the *authentication service* running as part of the platform on the device.

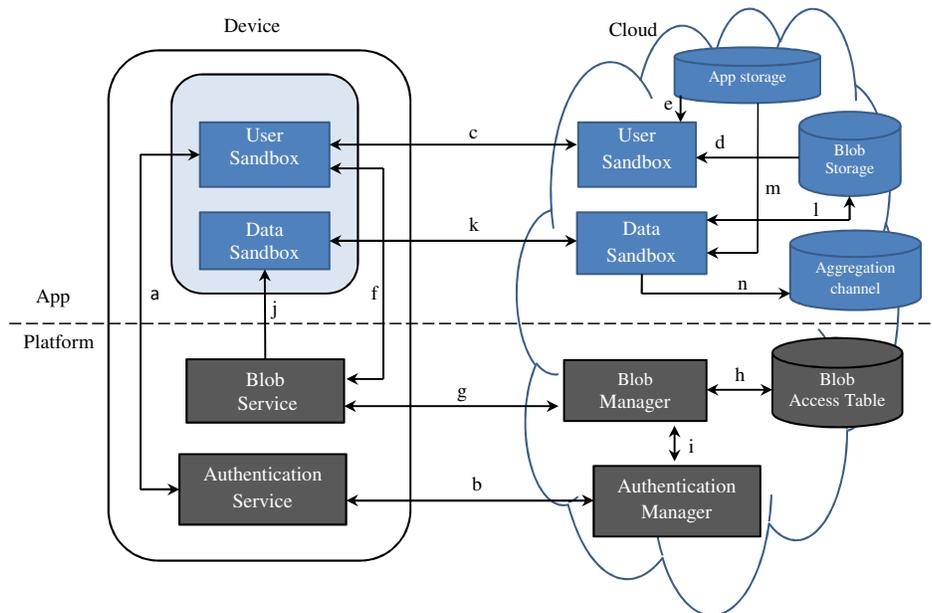


Figure 4.2: CleanRoom design. a - app invokes an authentication service. b - local authentication service talks to remote authentication manager. c - local user sandbox talks to remote half. d,e - read-only access (shown by unidirectional arrows) to user sandbox. f,g,h,i,j - control flow to start a data sandbox. k - local data sandbox talks to remote half. l,m,n - read-only (shown by unidirectional arrows) and read-write access (shown by bidirectional arrow) to data sandbox

This service sends user ID along with the apps's ID to the *authentication manager* (flow *b*) running as part of the platform in the cloud. Upon successful authentication, the manager starts up the requesting app's remote half for that specific user and opens a secure channel between the local and remote halves.

Once authenticated, the app spawns a viewer component, which runs inside a user sandbox and displays all the blobs this user has access to (flow *c,d,e*). A user may have multiple viewer components from different apps run-

ning inside a user sandbox. Since a user sandbox has read-only permission of the data, it cannot leak information outside of the sandbox.

Viewer component can do two things: it can create new blob, or it can read (but not modify) a blob for which this user has access. If the user wants to create a new blob, then the viewer component invokes (flow  $f$ ) *blob service* that runs as part of the platform on the device. This service takes user input such as name of the blob, who can access it, and in what capacity (such as read-write or read-only), and then allocates *blob storage* on the cloud. Each blob in blob storage is associated with the *owner* - whom enterprise trust with the data. The owner can either grant access or transfer the ownership of the blob to other users.

When a user chooses to edit a blob, the viewer component invokes blob service to spawn an editor component inside a data sandbox. Since the viewer component is untrusted, blob service confirms with the user the name of the blob being accessed. After user confirmation, it checks if this user has permission to edit the blob (flow  $g,h$ ). It then contacts the *authentication manager* (flow  $i$ ) to provide one time password and url of the cloud data sandbox which the device editor component can use to connect. Password is used to prevent an editor component from maliciously connecting to other cloud data sandboxes. CleanRoom then creates a secure channel between the device data sandbox and the cloud data sandbox and spawns the editor component on the device (flow  $j$ ). Editor component can then connect to cloud data sandbox (flow  $k$ ) using a one time password and url. User can now

edit the blob in the associated editor component.

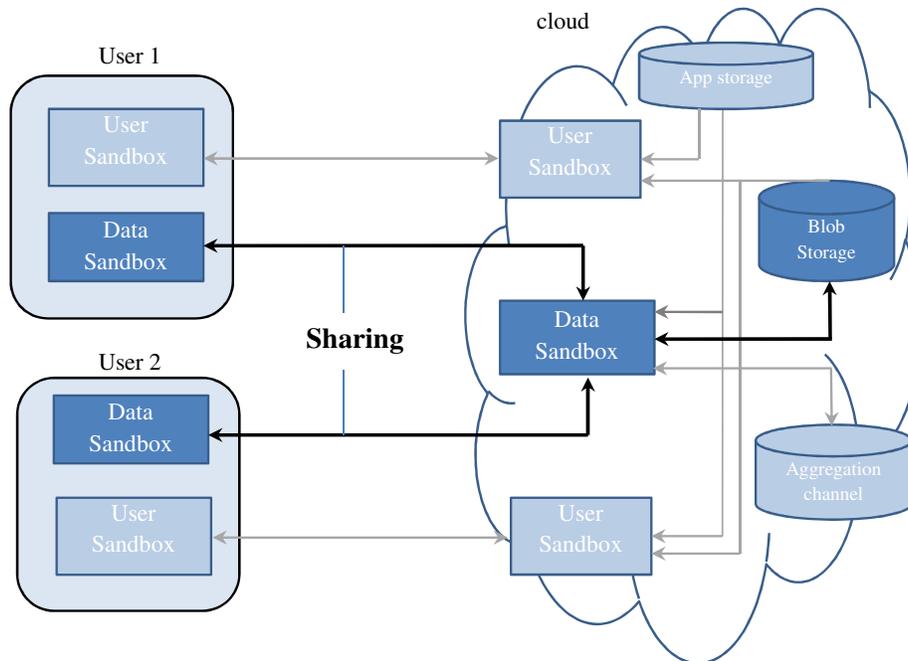


Figure 4.3: Sharing in CleanRoom

As shown in Figure 4.3, sharing or collaboration happens through *blob storage*, which can be accessed by multiple data sandbox components of the same app to edit it or can be read by multiple user sandbox components. CleanRoom does not interpret or process the layout or the content of *blob storage*

CleanRoom provides strong control over whom the content is shared with, but it has limited control on what an app shares. It is possible for an app to trick the user by indicating that it wants to share blob *A* but then ma-

liciously encode additional information other than what blob *A* originally had. CleanRoom does not handle this situation fully but it reduces the incentive of malicious app to do so as the encoded information is sent to a third-party outside of the control (and knowledge) of an app. Hence, improper sharing does not benefit the app developer directly. Moreover, such activities would raise suspicion about the activities of the app and are more likely to be caught. This concern is further discussed in Chapter 12

**Trusted clipboard.** Many a times a user may have to copy the data from one editor component to another. As apps components run in sandboxes, they are not allowed to arbitrarily communicate with each other. Therefore, CleanRoom provides a trusted clipboard in order to support the above functionality. So if a user wants to copy the content from one editor component to another, then she has to first copy the content to a trusted clipboard and then from that clipboard to the other component. In this process, CleanRoom presents the content of the clipboard to the user and confirms that the content is actually what she intended to copy. This process is not full proof as app may still trick the user but it atleast increases the chance of catching a malicious app.

**App to app communication.** Sometimes an app may need to communicate with another app. For instance, an email app may need to open a pdf attachment for which it needs to communicate with pdf reader app. If such communications are allowed without any control, it may leak information from

one app component associated with one set of users to another app component of a different set of users. CleanRoom handles this functionality by spawning the editor components of the different apps in the same data sandbox. So in the above example, when user wants to open the pdf attachment, CleanRoom will present the list of apps that can be used to read the pdf. The user can then select one of the apps and CleanRoom will spawn the editor component of the selected app in the data sandbox of email app, thus automatically associating the same set of users with the two app components.

**Exporting data outside the enterprise.** There may be instances when users may need to share data (e.g. company’s annual report) with people outside the company. Exporting and declassifying confidential data is a major challenge for any platform, including CleanRoom, as apps may leak information through hidden channels in the exported data (e.g., through steganography). CleanRoom currently addresses this issue by only allowing the export of human-readable data (e.g.text) from the data sandbox and prompting the user to confirm all exports, thereby making such leaks more conspicuous, limiting their bandwidth, and restricting them to the content of that blob alone. Such restrictions, which still allow some useful services (e.g., printing), preclude many other useful capabilities. Exploring these extensions are beyond the scope of this thesis and is left as a future work.

# Chapter 5

## Differential Privacy

*This chapter describes how CleanRoom uses differential Privacy while releasing aggregate statistics of an app to app publisher and quantifies the leakage through this channel.*

Apps often need statistics (e.g. ad impression count, most viewed articles) both for technical and financial reasons. Some apps use these statistics to improve users experience, while other use it to make money (by showing ads). Often aggregate statistics are sufficient. For example, some apps only need the information about most viewed article, and not whether individual user viewed the article. As stated earlier in section 4.2, only data sandbox components have a write access to aggregate channel and so information from a blob may still leak through these statistics. For instance, a malicious app can use a really high value in the statistics to leak sensitive information in a particular blob.

CleanRoom implements *Differential privacy* [27] to limit the amount of information leaked about a blob through app statistics. Informally, differential privacy is a framework for designing computations so that the influence of any single input on the output is bounded, regardless of the adversary's prior or

background knowledge or external sources of information it may have access to.

However, “conventional” differential privacy [27] which protects privacy in computations on a static data may not be directly applicable to report app statistics. Apps may continuously generate new data; for example, an app may continuously update the number of times a news article has been read. Therefore, new statistics must be periodically reported, and each report may leak additional information about a blob. CleanRoom uses a recently developed algorithms for *differential privacy under continual observation* [29] to support periodic release of statistics.

**Privacy-preserving counters.** In CleanRoom, the key building block for the aggregate channel is a set of platform-controlled *counters*. As app executes, its data sandbox components may increment one or more counters. At some time, the (randomized) value of these counters are released to the app publisher.

CleanRoom enforces *blob-level* differential privacy on these counters, i.e. the privacy of all data associated with a particular blob.

Formally, for some privacy parameter  $\epsilon$  (described later in this section), a computation  $F$  satisfies  $\epsilon$ -differential privacy if, for all (1) input datasets  $D$  and  $D'$  that differ only in inputs from a single data sandbox component which are present in  $D$  but not in  $D'$ , and (2) outputs  $S \subseteq \text{Range}(F)$ ,

$$\Pr[F(D) \in S] \leq e^\epsilon \cdot \Pr[F(D') \in S] \tag{5.1}$$

A standard mechanism for making any computation  $F$  differentially private is the *Laplacian mechanism*, which adds random noise from a Laplace distribution to the output of  $F$  before it is released:

$$F(x) + \text{Lap}\left(\frac{\Delta F}{\epsilon}\right)$$

Here,  $\text{Lap}(y)$  is a Laplace-distributed random variable with mean 0 and scale  $y$ , and  $\Delta F$  is the maximum possible change in the value of  $F$  which is referred to as  $F$ 's sensitivity when inputs from a single data sandbox component are removed from the dataset.

Intuitively, the more sensitive a computation is to its input, the more random noise is needed to ensure a given level of privacy. Therefore, the amount of noise that CleanRoom adds to the released counter values depends on the number of counters a data sandbox component can update ( $n$ ) and the maximum amount by which a data sandbox component can affect any single counter ( $s$ ). There is a trade-off in the Laplacian mechanism between privacy and accuracy: *higher accuracy requires giving up more privacy*.

**Supporting periodic updates.** Many apps need to periodically release updated statistics on app usage. The Laplacian mechanism can be applied for every release of the counters, but for infrequently updated counters, the random noise added by the mechanism can be significantly larger than the counter's value (which may be very small), resulting in high relative error.

```

1:  $V_i$  : true count in period  $i$ 
2:  $\lambda \leftarrow \frac{s \cdot n}{\epsilon}$ 
3:  $A \leftarrow 0$ 
4:  $D \leftarrow b + Lap(\lambda)$ 
5: for each period  $i$  of duration  $1/f$  do
6:    $A \leftarrow A + V_i$ 
7:   if  $A - D > Lap(\lambda)$  then
8:     Release  $A + Lap(\lambda)$ 
9:      $A \leftarrow 0$ 
10:     $D \leftarrow b + Lap(\lambda)$ 
11:   end if
12: end for

```

Figure 5.1: Algorithm for delayed-output counter.

To solve this problem, CleanRoom uses *differential privacy under continual observations* [29] to provide *delayed output counters*. Figure 5.1 describes how such a counter is implemented. Intuitively, if the value of the counter is small relative to the noise that must be added in order to release it, the release is randomly delayed. Furthermore, CleanRoom only releases counters on a fixed schedule (line 5). Even if a counter has accumulated a large number of updates, it may not be immediately released. Delaying the release of counters may affect the freshness of the released values, but the benefit is that these values are less noisy and thus more accurate.

**Choosing privacy parameters.** Achieving absolute privacy while providing useful information based on private data is impossible. Dinur and Nissim showed that even if answers to queries about a private dataset are perturbed by so much random noise as to render them essentially unusable, the entire

dataset can be reconstructed after a linear number of queries [26]. This means that each output, no matter how noisy it is, leaks some information about the private data.

To model the cumulative loss of privacy after multiple computations on the same private data, differential privacy uses the notion of a *privacy budget* [28, 42]. Every  $\epsilon$ -differential private computation charges  $\epsilon$  cost to this budget. Intuitively, the higher the value of  $\epsilon$ , the less noise is added, the more accurate the released value, but the privacy cost is correspondingly higher. Once the privacy budget is exhausted, no more computation is allowed.

Unfortunately, above approach cannot be used as it is, since apps are usually expected to run for long periods of time and it is undesirable that an app loses its functionality after sometime. Instead, CleanRoom enforces a *per-period* privacy budget that bound *privacy load per period* by parameter  $R$ , which is chosen by the platform. For a given  $R$ , the app publisher may specify parameters in Table 5.1 so long as

$$\epsilon \times f \leq R \tag{5.2}$$

One way to interpret  $\epsilon$  is in “Bayesian” terms. If  $P$  is the adversary’s prior probability about a blob having a particular value and  $P'$  is the posterior probability (after observing the output), then differential privacy guarantees that:

$$P' \leq e^\epsilon \times P$$

Principal	Parameter
Platform provider	Per-period privacy budget ( $R$ )
App publisher	List of counters ( $L$ ) Frequency of output release ( $f$ ) Privacy parameter ( $\epsilon$ ) Max. number of counters single user can update each period ( $n$ ) Max. value a user can contribute to each counter each period ( $s$ ) Buffer size* ( $b$ )

Table 5.1: Parameters for counters.

regardless of the actual adversarial prior. If the adversary’s uncertainty about the blob is measured as min-entropy of his probability distribution over the possible values of the data, then the adversary’s information gain would be expressed as  $(\epsilon \log_2 e)$  [18, 21]. Given this representation of uncertainty, CleanRoom’s counters release at most  $(f \cdot c \log_2 \epsilon) = (R \log_2 e)$  bit per period. For example, with  $\epsilon = 1$ , and release frequency  $f =$  once per day leaks at most 1.44 bits of information daily.

As long as condition (5.2) is satisfied, an app publisher is free to choose the value for the parameters listed in Table 5.1. For example, the app publisher may decide to output more frequently ( $f$ ), at the expense of lower accuracy, decreasing the maximum number of counters ( $n$ ), the maximum amount they can contribute ( $s$ ), or some combinations of these values.

# Chapter 6

## Information Flow Model

*This chapter describes an information flow model of CleanRoom and shows that the model enforces the desired enterprise privacy policies.*

IFC model of CleanRoom tracks information flow between *entities*. Entities are of two types - principals and objects. *Principals* represent either a human or running code such as user, platform, user sandbox, data sandbox, or external source. These are entities that act in the model. On the other hand, aggregation channel, blob storage, private vaults and app storage are *objects* and are acted upon by the principals.

The IFC model of CleanRoom has been adapted from previously proposed models [39]. Like earlier models, it uses tags and labels to track information flow between entities in the system. A *tag* is an immutable, unique identifier associated with an entity. It can be a string or a numeric value. For example, tags  $as_i, ac_i, us_{ij}, ds_{ik}$  are associated with app storage, aggregation channel, user sandbox, and data sandbox respectively, where subscripts  $i, j, k$  represent a particular app, user, and data sandbox respectively. Since private vaults and blob storage are tied with their sandboxes, they are modeled as part of the sandboxes. For example, private vault of user sandbox receive a

tag of user sandbox. Similarly, blob storage receive a tag of associated data sandbox. Platform has a tag  $\top$  and users have a tag  $u_i$ . The model uses a tag  $\perp$  to represent all untrusted external sources such as public files, external servers, among others. Together these entities form a privilege hierarchy shown in Figure 6.1.

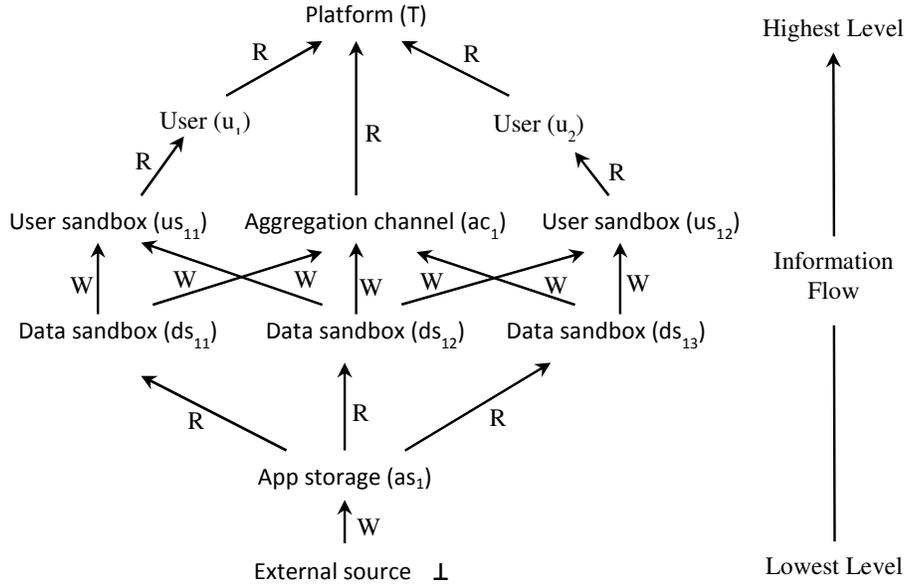


Figure 6.1: R - a higher level entity can *read* from a lower level entity. W - a lower level entity can *write* to a higher level entity. Information flow is allowed from lower level to higher level but not in other direction.

A principal at a particular level in the hierarchy can read from lower level and can write to objects at a higher level (no read-up, no write-down). If an entity has no link to another entity in the privilege hierarchy, then the two entities cannot interact with each other. Below are the reasons for placing

various entities at a particular level in the privilege hierarchy:

- Untrusted external sources are at the lowest level because they can write to app storage but cannot read from any entity of the system.
- App storage is placed higher than the untrusted external sources because it can receive information from external sources which can be read by user and data sandboxes of an app.
- Data sandbox can read from app storage and can write to its private vault and aggregation channel. It can also have one way communication with user sandbox such that it can send information to user sandbox but cannot receive information from it. So it is placed above app storage and below aggregation channel and user sandbox in the hierarchy.
- User sandbox and aggregation channel are at the same level and above the data sandbox. This is because the user sandbox can receive information from a data sandbox but cannot write to aggregation channel.
- User has higher privilege than any other entity and can change the privileges of lower level entities. For example, if user  $A$ , who is the owner of a *blob*  $d$ , gives read access to another user  $B$  then privileges of user sandbox of user  $B$  changes such that it can access *blob*  $d$  now.
- The platform is at the highest level of privilege hierarchy. It is higher than the user because it enforces enterprise policies and can allow or deny access of a blob to a user. For example, according to an enterprise

policy, salary document of an employee  $A$  should be accessible only to  $A$  and employees in the accounts department. In this case, platform will deny access to all other users except  $A$  and employees in the accounts department.

The platform maintains an access control list with each blob which contains information about the owner of the blob, the set of users with read access, and the set of users with write access to the blob

A label  $L$  is a set of tags. Unlike tags, labels are mutable and participate in information flow control. Labels are assigned by platform in accordance with the access control list and privilege hierarchy. The label assigned to an entity, at a particular level of privilege hierarchy, has its own tag and tags of all the lower levels entities. For example, label on a user sandbox is  $L_{us_{11}} = \{us_{11}, ds_{11}, as_1\}$  which consists of tags of user sandbox ( $us_{11}$ ), data sandbox ( $ds_{11}$ ) and app storage ( $as_1$ ). Since  $\perp$  is part of every label, it is omitted for a concise label expression. Below are some sample labels on various entities of an app:

- App storage has a label  $L_{as_1} = \{as_1\}$
- Aggregation channel has a label  $L_{ac_1} = \{ds_{11}, ac_1, as_1\}$
- Data Sandbox has a label  $L_{ds_{11}} = \{ds_{11}, as_1\}$
- User Sandbox has a label  $L_{us_{11}} = \{us_{11}, ds_{11}, as_1\}$

## 6.1 Secrecy

The platform enforces information flow control through labels. Information can flow from source to destination if the label on the destination ( $L_d$ ) is more restrictive than the label on the source ( $L_s$ ), where restriction is defined as  $L_s \subseteq L_d$ . Consider an example label on data sandbox as  $L_{ds_{11}} = \{ds_{11}, as_1\}$  and user sandbox as  $L_{us_{11}} = \{us_{11}, ds_{11}, as_1\}$ .

Then data sandbox can:

- read from app storage  $L_{as_1} = \{as_1\}$  and  $L_{as_1} \subset L_{ds_{11}}$
- send data to user sandbox  $L_{us_{11}} = \{us_{11}, ds_{11}, as_1\}$  and  $L_{ds_{11}} \subset L_{us_{11}}$
- write to aggregate channel  $L_{ac_1} = \{ds_{11}, ac_1, as_1\}$  and  $L_{ds_{11}} \subset L_{ac_1}$

Similarly, user sandbox can:

- read from app storage  $L_{as_1} = \{as_1\}$  and  $L_{as_1} \subset L_{us_{11}}$
- receive data from data sandbox  $L_{ds_{11}} = \{us_{11}, ds_{11}, as_1\}$  and  $L_{ds_{11}} \subset L_{us_{11}}$
- can not write to aggregate channel  $L_{ac_1} = \{ds_{11}, ac_1, as_1\}$  and  $L_{us_{11}} \not\subset L_{ac_1}$

**Control flows.** Figure 4.2 shows all the control flows that happen for an app running in CleanRoom. The IFC model of CleanRoom ensures that information leak does not happen through any of these control flows. According to information flow rules, user and data sandbox components can only read the data (flow  $e,m$ ) from the app storage but cannot write to it. So information cannot flow out of the system through app storage. Besides, only data sandbox components are allowed to write to an aggregation channel (flow  $n$ ). Since a user sandbox may have information from different apps of the same user, restricting the access of user sandbox to aggregation channel prevents information flow from different apps. Moreover, information from aggregation channel is accessible to external source only after it is sanitized (via differential privacy) by the platform. Therefore, the information leak through aggregation channel is bound by differential privacy.

When a user accesses a particular blob or creates a new blob, control gets transferred from a user sandbox component to the specified data sandbox component (flow  $f,j$ ). If a user sandbox component is allowed to create any blob then the user sandbox component may use the name of blob to leak confidential information to data sandbox component. For example, a user sandbox component may create blob with name XYZ based on information in blob  $A$  to which this user sandbox has read access. Thus, leaking information from blob  $A$  to blob XYZ. CleanRoom prevents such information leaks by obtaining the name of the blob from the user. Moreover, each invocation of data sandbox component (for already created blob) is confirmed from the

user, preventing user sandbox component to arbitrarily invoke a data sandbox component.

Information from a data sandbox component is allowed to flow to a user sandbox component of the same user (flow  $k,l,d,c$ ). Besides, a user sandbox and a data sandbox component are only allowed to connect to their corresponding cloud half (flow  $c,k$ ).

**Other system resources.** CleanRoom does not allow user or data sandbox component to create world-readable files. This prevents information leak through file system. Moreover, CleanRoom provides a rough estimate of time to app components, preventing these component to leak information through timing channels. Also, these components are not allowed to establish arbitrary network connection to a third party.

## 6.2 Case Study: Text Editor app

Consider users *Bob* and *Alice* who are employees of a company X and use a Text editor app to collaborate on a project *A*. However, *Bob* has some documents from project *B* which he does not want *Alice* to access. Below is the description of how CleanRoom helps *Bob* to collaborate with *Alice* but prevents *Alice* from accessing documents that she does not have access to.

Initially, the platform will create two user sandboxes with labels  $L_b \in \{us_b, as\}$ ,  $L_a \in \{us_a, as\}$  for *Bob* and *Alice* respectively. At this point the user sandboxes of each user can read app storage to display any app publisher

provided document such as README. Now, *Bob* creates two documents, first one for project *A* and another for project *B*. *Bob* shares document belonging to project *A* with *Alice*. Platform will create two data sandboxes with labels  $L_A \in \{ds_A, as\}$ ,  $L_B \in \{ds_B, as\}$  for documents of project *A* and *B*, respectively. It will also update the labels on user sandboxes of *Bob* and *Alice* as  $L_b \in \{us_b, ds_A, ds_B, as\}$ ,  $L_a \in \{us_a, ds_A, as\}$ . With this labeling CleanRoom IFC model achieves the desired privacy policy for *Bob*. *Alice* obtains access only to data sandbox of the shared document. User sandbox of *Alice* can access *Bob*'s document from project *A*  $L_A \subseteq L_a$  but cannot access *Bob*'s document from project *B*  $L_B \not\subseteq L_a$ . Since *Bob* is the owner of the shared document, it can add another user *Charlie* to the document for project *B*. Now, *Charlie* will obtain access to data sandbox of project *B*. *Charlie*'s user sandbox will have label  $L_c \in \{us_c, ds_b, as\}$ . If *Bob* wants to transfer the ownership to *Charlie* then platform will update its access control table and replace *Charlie* with *Bob*

### 6.3 Characteristics of the model

Following are the characteristics of the model that differentiate it from earlier IFC models:

- It has been applied at the granularity of CleanRoom abstraction (such as data sandbox, user sandbox, storages, and communication channels) instead of at the language level. Although language level provides fine-grained information flow control, it makes the model complex because of implicit and explicit information flow issues. Besides, programmers

have to learn a new language and have to annotate their code to specify flow policies which is tedious and error prone process [19].

- *Predefined security class hierarchy instead of per-app decentralized policies.* CleanRoom has centralized policy which reflects the enterprise privacy policies and has only four levels of class hierarchy. This implies that data in CleanRoom flows only to a short distance making it easier to track the information.
- *Narrow interface instead of wide.* CleanRoom restricts the application environment and allows communication only through platform controlled storage and communication channels which allow information to be tracked at the level of these abstractions.

Having said that, the model is concerned only with privacy and not with other aspects of security (integrity and authenticity). It is possible that addition of these concepts would make the model complex. Another aspect is the flow of information through side channels (e.g. timing channel) which is not handled by this model.

# Chapter 7

## Implementation

*This chapter describes in detail the implementation of CleanRoom .*

### 7.1 Background

CleanRoom prototype is built using an Android [1] platform on the device side and Jetty webserver [10] on the cloud side. It uses Hbase [6] for its cloud storage. This section briefly discusses these resources which will help later in understanding the implementation of CleanRoom.

#### 7.1.1 Android

Android is an open-source operating system for mobile devices such as smartphones and tablets. It is based on Linux and provides base operating system, an app middleware layer, and a collection of system apps. It is designed keeping security and protection of apps in mind. This section presents a description of the design of Android that provides basic security from malicious apps.

Apps on Android consist of multiple components. Each component is identified by its type. An app developer chooses the type of a component from

predefined types depending on the purpose of the component. Android has four component types:

- *Activity* component is used for user interface. An app developer can design different user interfaces for different purposes but at any time only one user interface is active and has keyboard and screen control. An activity can start another activity, and can exchange values with it. One of these activities is the main activity, specified in the manifest file of an app and appears as a first screen on the start of an app.
- *Service* component is used for two purposes, one is to carry out any background processing that an app may need to perform such as download a file from remote server and the other, is to provide functionality to other apps. In this case, service runs as an app-specific daemon and provides a RPC (Remote Procedure Call) interface that other apps can use to receive and send data.
- *Content Provider* component stores app specific data. Each content provider has an associated authority name which is used as a handle by other components to read and write content. Content is usually stored as a relational database, but it can also be stored as a file.
- *Broadcast receiver* components is used to receive messages from other applications. An app can dynamically register an instance of broadcast receiver or statically publish it in the manifest file. An app can broad-

cast messages by including the namespace assigned to the destination application.

App components interact with each other via *intent*, which is simply a message object with a destination address and data. This object is passed to Android provided APIs, which carry out the necessary inter component communication (ICC). This communication mechanism among the components is same, irrespective of whether a component belongs to same or different apps.

**Android Security.** Android provides security through two mechanisms , one by running each app as a different user identity, leveraging underlying Linux system security, and other by having a reference monitor at the ICC level. Because of this design, one app cannot directly affect the functionality of another app. Android reference monitor mediates all ICC establishments and enforces mandatory access control based on the labels assigned to apps and their components. Labels are the collection of access permissions assigned by the app developers and specified in the app manifest file. Labels are immutable and can only be changed via re-installation of the app. When a component initiates a communication with another component, then reference monitor checks the label of the target component in the collection of labels assigned to its containing app, and if the label is present then reference monitor allows the establishment of this communication, otherwise drops it. This mechanism is enforced even if both the component belongs to the same app. Android

only restricts access to a component via mandatory access control, it does not provide any information flow guarantees between the two apps.

### 7.1.2 Jetty WebServer

CleanRoom uses Jetty webserver to provide cloud side isolation. Jetty is an open source, Java-based web server. It provides security at two levels, one by placing each webapp in a separate web context which restricts a webapp from accessing the content of another webapp, and other through Jetty Policy which integrates the security mechanisms of Jetty with the core JDK security. A typical Jetty Policy in CleanRoom looks like:

```
grant codebase "<path of codebase>" {  
    permission java.security.AllPermission;  
}
```

Jetty Policy enables an administrator to specify permissions to a webapp. These permissions could be either to allow a webapp to read or write a file or to restrict its access to other system resources. The policy for a webapp is represented by a JettyPolicy object which is configured through the policy files specified by the administrator. At the start of a webapp, the JettyPolicy object is loaded so that all security sensitive actions can be taken. In order to access a resource, the entire call stack is checked to validate that every object in the stack has the permission to access the resource. Permissions are granted to a source code, which is specified in the policy file either as a specific location on the disk or as location of directory containing jar and class files.

### **7.1.3 Hbase**

CleanRoom storage system uses Hbase as a cloud storage system. Hbase is an open source, non-relational, distributed database. It is a column-oriented database that runs on top of HDFS (Hadoop Distributed Filesystem). It does not support structured query languages such as SQL and is mainly suitable for sparse data sets. Hbase consists of a set of tables. Each table is made up of rows and column. Multiple columns can be combined into a single column family, which is then stored together. Like a relational database, lookup in the table happens via a primary key. A user of Hbase has to specify the schema of the table before creating it, however, more columns can be added to a column family at run time. Underneath Hbase is HDFS which has a name node and region server. Name node controls the distribution of the tables whereas region server stores a portion of the table and operates over it. Like Android and Jetty, applications for Hbase are also Java based.

## **7.2 CleanRoom Implementation**

CleanRoom prototype is build using Android 2.3 (Gingerbread) for the mobile device, Jetty for the cloud services, and HBase for the cloud storage and communication channels. The trusted computing base (TCB) consists of the above software, cloud operating system (e.g. Linux), and CleanRoom implementation. In terms of code size device side implementation is about 2,700 lines of code and cloud side is 7,500 lines of code. The design of CleanRoom is largely agnostic to the specific sandboxing technology and could have

used virtual machines, Native Client [12], or more advanced sandboxes, which would change the size of the TCB.

### 7.2.1 Isolations and authentication

CleanRoom provides mechanisms for sandboxing the app components regardless of where they are running. On the device side, user and data sandbox app components are implemented as two separate Android apps. This section explains how CleanRoom provides isolation and controlled communication between these app components.

**Isolation on device.** To implement the sandbox on the device, CleanRoom augments Android’s built-in sandboxing mechanism. By default, Android assigns each app a unique user identifier (UID). CleanRoom allows non-privacy-preserving apps to coexist with privacy-preserving apps on the same device, but assigns UIDs from different ranges to apps of different types. CleanRoom assigns privacy-preserving apps UIDs 10001 to 30000, and legacy non-privacy-preserving apps UIDs 30001 to 60000. User and data sandboxes are implemented as two separate apps, and are assigned different UIDs. However, CleanRoom internally maintains a map of UIDs, assigned to user and data sandboxes of the same app. This makes isolation enforcement simpler in the kernel code as apps and their sandboxes can be differentiated based on UIDs.

Android uses standard Linux permissions to isolate apps from each other, but this is not enough to prevent an app from abusing the permissions

it has. In particular, Android apps can:

- create world-readable files that can be read by other (potentially malicious) apps or write to files owned by and hence readable by other apps;
- use Android's IPC mechanisms to communicate private information to other apps; or
- perform arbitrary network transmission and transmit private information to other parties.

To prevent leaking of private data through the above channels, CleanRoom modifies Android to block apps from creating world-readable files or directories, and from writing to files or directories owned by another app's UID. This implies that an app can only write to directories that it alone has read access to and that other apps cannot see the files it has written to. CleanRoom does not allow confined apps to communicate with other non-system apps via IPC, including Binder IPC (the basic primitive for various higher-level Android IPC mechanisms) except from app representing data sandbox to app representing user sandbox of the same user. As mentioned above, CleanRoom, through UIDs, can easily find whether the data and user sandboxes belongs to the same app. Finally, CleanRoom uses iptables to confine the apps' network traffic. These changes are only applied at the kernel level to CleanRoom-confined apps (recognized by their UIDs).

**Isolation on cloud.** CleanRoom implements the server-side functionality for apps as Java servlets using Jetty. Many existing Web apps, e.g., those on Google App Engine [4], can thus be easily adapted to CleanRoom.

In Jetty, each app is isolated in a separate Web app context, a container that shares the same Java class loader. Although separate contexts provide isolation across different web apps, different instances of the same web app (corresponding to different users accessing the same Web app), share the same classloader and can access other users' Java objects and communicate through static members. CleanRoom eliminates these shared states by creating a separate web app context for every servlet instance (belonging to user and data sandboxes). This effectively implements isolation at the level of the Java classloader and prevents one user's Java objects from accessing other user's Java objects or state. Besides, a user's user sandbox servlet cannot access the objects of the user's data sandbox servlet. Moreover, each user of an app is also isolated in a separate context, achieving classloader-level isolation.

To restrict the servlet's communication via system resources, CleanRoom relies on Java's security monitor. For example, a policy specifies that servlets can neither create network sockets nor access local files except in a scratch directory that is exclusively assigned to a particular per-app, per-user sandbox. This scratch directory facilitates the use of existing libraries, which require accesses to a file system for generate temporary files. Besides, CleanRoom sandbox also includes many other restrictions used by Google App Engine, e.g., disallowing reflection and controlling access to JVM-wide resources

such as system properties.

**Authentication.** When an app on the user’s device wants to communicate with its cloud counterpart, it sends an “intent” to CleanRoom’s local trusted authentication service, implemented as a system app. After identifying the requesting app, the authentication service requests for the user’s credentials via user input or from a cache and sends them, along with the app’s ID, through a TLS tunnel to CleanRoom’s authentication manager in the cloud. Upon successful authentication, the authentication manager sets up a new servlet instance at a specific URL, establishes an IPsec endpoint on the machine where the servlet is instantiated, and sends this URL, a one-time password that is required to access the servlet instance, and the IPsec key to the authentication service on the user’s device.

The authentication service establishes the other end of the IPsec tunnel on the device, updates iptables to allow the app to communicate with the servlet, and passes, via intent, the URL and password to the app. IPsec ensures that all communication to and from the servlet is encrypted, and iptables ensure that the app on the user’s device can only communicate with the user’s servlet instance via this IPsec tunnel. Finally, the app running locally on the user’s device authenticates using the provided password via HTTP basic authentication over the IPsec tunnel (which encrypts the credentials); this step ensures that only this particular app can communicate with the servlet. Once this process is complete, the app can send HTTP requests to the provided

URL and receive HTTP responses from its cloud component.

### 7.2.2 Storage and communication channels

CleanRoom’s storage systems use local device storage and HBase. Local device storage is part of CleanRoom’s private vault. Both user and data sandboxes have their own private vault. Any data that is written to local storage is secured as described in Section 4.2 and cannot be exported from the sandbox. Access to cloud storage is provided via a HBase-like API.

When an app publisher submits an app to the platform, the publisher provides a WAR (Web application ARchive) file that contains the app’s servlet code and XML files that describe the schema of the HBase tables that the app needs for each type of cloud storage. To implement various channels, CleanRoom provides wrappers of the HBase client that expose the appropriate interfaces to servlet instances. For example, the interface to content storage exposes read-only operations on the storage’s shared tables. The interface to the cloud-backed blob storage provides both read and write access to data sandbox and read-only access to user sandbox of the same user. The wrapper for the aggregate channel exposes an update-only interface to data sandbox for the counters, which are stored in the HBase tables by CleanRoom. Stored counter values are periodically released by (1) sanitizing them via the differential privacy module (Chapter 5) and (2) writing them to a table that can be read by the publisher.

# Chapter 8

## Applications

*To demonstrate that useful business apps can be built under CleanRoom, this chapter presents three representative apps that are developed on CleanRoom.*

### 8.1 Text Editor

Text Editor is a prototype android app in which a user (employee) can create and share documents to collaborate with other users. The *viewer* and the *editor* components are implemented as two separate apps on device side and as two different servlets on the cloud side. Even though viewer and editor are two separate apps, they give an impression of single app to the user. For example, there is only one app icon shown to the user. At the time of app installation, CleanRoom obtains the name of the two apps along with their type (viewer or editor) from the manifest file and enforces privacy rules between the two apps. Viewer component is to display a list of private and shared documents which allows a user to select and read any document from the list. It also allows a user to create new documents. On the other hand, editor component is to modify a document. Text Editor defines each document

as a blob and so CleanRoom provides privacy guarantees for each document.

Figure 8.1 shows the various control flows that can happen in the Text Editor app. When a user opens the Text Editor, the platform spawns the app's viewer component in a user sandbox that is specific to this user. This component, which has read-only access to all user-accessible documents, lists all the documents and relevant metadata. Any customizations that a user prefers, e.g., sorting by name, can be stored and retrieved from the user sandbox's private vault. User can take three action in this sandbox:

1. create a new document
2. change the setting of the document such as share the document with other users, list all the users with access to this document etc.
3. read any listed document

When a user wants to create a new document, the viewer component invokes platform provided *blob service* and control gets transferred from user sandbox to blob service. Blob service then directly interact with the user to obtain the name of the document and the set of other users who have read-only or read-write permission to this document. Similarly, blob service is invoked in case when user wants to change settings of existing document. Blob service interact with the user to obtain the new settings (e.g. sharing the document with a new user). If the user just wants to read one of the listed documents then the app's viewer component sends a request to user servlet running in the cloud to fetch the content of requested document.

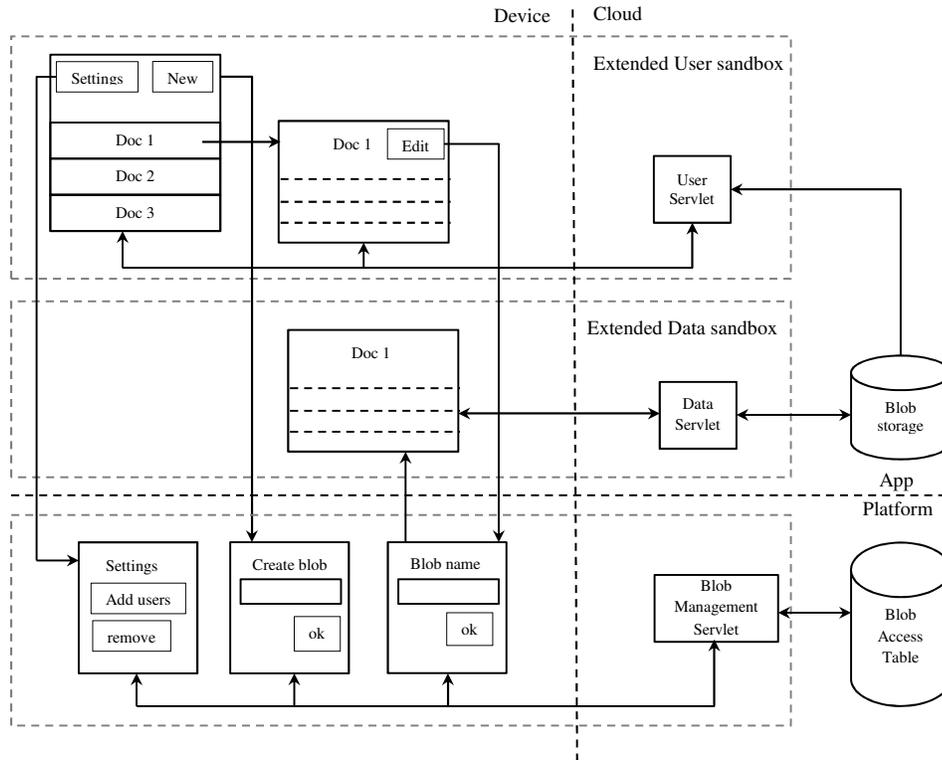


Figure 8.1: Text Editor Design

When a user wants to modify a document, the app's viewer component sends a request to blob service to open the selected document. Blob service prompts the user about the choice of document and upon receiving user confirmation, blob service verifies if the user has permission to modify the document. In case the user has permission to modify the document, blob service starts data servlet on the cloud for this document if it is not already running, generates a one time password and url to connect to the cloud data sandbox, and spawns the app's editor component in the data sandbox on the

device. It then passes the one time password and url to the editor component to connect to the cloud data sandbox. Viewer component then fetches the content of the document and allows the user to edit the document. In case the user does not have access to the document, blob service silently returns to the user sandbox. Finally, any customizations specific to the user may also be stored in the data sandbox's private vault.

Figure 8.2 shows a few screenshots of Text Editor application. As mentioned above, viewer component displays a list of documents that the user has created earlier. In the viewer component, the user can either read any of the listed document or can create a new document. Either of these actions are intercepted by the platform provided blob service which confirms the action with the user and also obtains the necessary inputs, like the name of the document in case user wants to create a new document. Blob service then spawns the editor component of the app in a data sandbox which allows the user to edit the document.



Figure 8.2: Text Editor application screenshots

## 8.2 Calendar

Calendar app allows its users to view, create, and edit events on a calendar, create new calendars, and share calendars with other users. In CleanRoom sharing happens at the granularity of a blob so calendar app defines each calendar as a blob and is stored in cloud-backed blob storage that allows its retrieval from multiple devices. The Calendar's viewer component has read-only access to all user-accessible calendars, which allows the user to see all the events. Calendar app creates a user view that includes events from all the calendar into one calendar. Besides, it also shows a list of shared and private calendars. As CleanRoom provides privacy to each calendar (blob), each calendar is associated with an owner, a set of users with read permission and a set of users with write permission.

When a user opens the Calendar app, the platform spawns the app's viewer component in a user sandbox that is specific to this user. A user can take three action in this sandbox:

1. customize its view of calendars and events
2. change the setting of a calendar, or
3. create a new calendar

A user action to customize view of calendars and events is handled by the viewer component. Any change in view is stored in user sandbox private vault for future reference. The mechanism of creation of a new calendar or

change in setting is exactly same as described in Text Editor app. When a user wants to edit a calendar (remove or add an event), then the app's viewer component sends a request to blob service to open the selected calendar for edit. Blob service confirm this action with the user. Upon confirmation and if the user has write permission to the calendar, it then starts the cloud data sandbox servlet for the calendar if it is not already running, generates a one time password and url to connect to the cloud data sandbox, and spawns the app's editor component in a data sandbox on the device. It then passes the one time password and url to the app's data sandbox to connect to the cloud data sandbox. User can then edit the calendar in the spawned data sandbox.

Figure 8.3 shows a few screenshots of Calendar application. As mentioned above, viewer component displays a list of calendars that the user has created earlier. App developer can implement a more fancier interface where events from all the calendar can be shown in one calendar with different colors. In the viewer component, the user can either add more events to existing calendars or can create a new calendar. Either of these actions are intercepted by the platform provided blob service which confirms the action with the user and also obtains the necessary inputs, like the name of the calendar in case user wants to create a new calendar. Blob service then spawns the editor component of the app in a data sandbox which allows the user to edit the calendar.



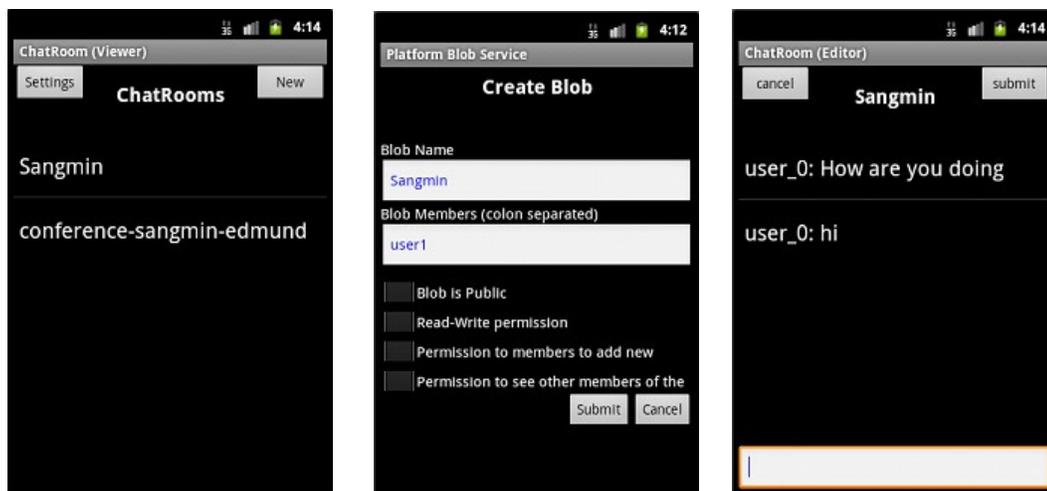
Figure 8.3: Calendar application screenshots

### 8.3 Instant Messaging

Instant Messaging is an example of a chat app where sharing happens when two or more users interact with each other. Every contact in a user's contact list is considered as a chat room. Instant Messaging app models each chat room as a blob which stores the contents of the conversation that the user had with the contact. By displaying a list of all chat rooms, the viewer component is effectively displaying a user's contact list. A user can create new chat rooms by invoking blob service which interact with the user to get the name of the contacts. A chat room can have more than one user. The creator of the chat room is the owner of the chatroom and can add other users in the list of authorized users or she can transfer the ownership of the chatroom. When a user selects a particular contact to interact with, the platform spawns the appropriate editor component specific to the contact which then access the blob storage for messaging and logging functionality. Since chat is an interactive app which needs constant update/notification for new messages, editor component on the device maintains two connections with the cloud data sandbox servlet, one to send chat messages and other to poll for new messages. Cloud servlet holds the request for new messages till it receive some message from other user. Once it receive a new message it iterate through the list of requests on hold and send them the new message. After receiving the new message the editor component send another poll request. This process continues till users opt out of the chat room.

Figure 8.4 shows a few screenshots of Instant Messaging application.

As mentioned above, viewer component displays a list of chatrooms that the user has created earlier. These chatrooms can either be a single contact or a group of contacts (for conference chat). In the viewer component, the user can either access any of the listed chatrooms or can create a new chatroom. Either of these actions are intercepted by the platform provided blob service which confirms the action with the user and also obtains the necessary inputs, like the name of the contact in case user wants to create a new chatroom. Blob service then spawns the editor component of the app in a data sandbox which allows the user to chat with the contact.



List of ChatRooms  
(Viewer component)

Create a New chatroom  
(Platform Blob service  
obtains inputs from the user)

Chat  
(Editor component)

Figure 8.4: Instant Messaging application screenshots

# Chapter 9

## Design Guidelines

*This chapter presents few design guidelines that will help app developers to develop apps on CleanRoom*

**Unit of sharing.** In CleanRoom, the unit of sharing is referred to as a blob. This general abstraction can be used by an app developer to specify unit of sharing for the app. For example, in the text editor app a blob is a text file, calendar app defines each calendar as a shared blob, and instant messaging app uses conversations with a friend as a blob. Since CleanRoom provides privacy guarantees at a level of a blob, an app developer can decide the granularity of sharing based on its privacy requirements. For instance, if a calendar app wants to provide privacy at a level of an event in the calendar, then each event becomes a blob and sharing would then happen at an event level. An email app can manage privacy for every single email, in which case an email will be a blob or can provide privacy for all emails exchanged with a particular contact, in which case all the emails exchanged with that contact will become a blob. CleanRoom does not interpret the content of a blob and leaves it to app developer.

**Viewer component.** A viewer component is an app component that runs inside a user sandbox. CleanRoom confines a user sandbox such that data is allowed to flow in from other entities (editor or storage) but can only flow out to trusted entities (such as clipboard). An app developer can use this abstraction to provide a customized, read-only interface for blobs. For example, the text editor app can organize text files into various folders such as private, shared and work related, and provide an interface to read them. A calendar app can overlap different calendars into one calendar and show events from different calendars with different colors.

**Editor component.** An editor component is an app component that runs in a data sandbox. CleanRoom confines a data sandbox such that data can only flow in from app storage and private vault of this data sandbox and it can flow out to aggregation channel and user sandbox. An app developer can design an editor component to handle updating a blob that is shared. Since CleanRoom does not allow sharing a blob from user sandbox, an app necessarily needs to have a data sandbox component if it supports some sort of data sharing. However, data sandbox component is optional in case when an app does not have any sharing feature. For example, a book reader app need not have an editor component if the only interface it provides is just to read a book available through app storage.

**Cloud and user information.** CleanRoom provides APIs to an app to fetch information about the cloud and its users. An app can use this infor-

mation to connect to remote cloud or display information about a user. For example, an instant messaging app uses user name to show messages from different users.

# Chapter 10

## Evaluation

### 10.1 Performance overhead

The performance of CleanRoom was evaluated using a server with two four-core Xeon E5430 CPUs and 16 GB RAM and 4 clients with a single-core 3 GHz Pentium 4 Xeon CPU with hyper-threading and 1 GB of RAM, all running Fedora 8.

The throughput and response time of the various mechanisms employed by CleanRoom were measured using two types of workloads: a simple static workload where server responded with about 10 bytes of static HTTP data, and a computationally intensive workload where the server randomly generates 1 MB of data and calculates its SHA-256 hash. The workloads were generated by having a varying number of clients continuously submit requests over a 30-second interval.

Figure 10.1 shows the results with different mechanisms turned on. In the base configuration, Java security monitor was disabled, no isolation (i.e. a single servlet instance served all client requests), and without an IPsec tunnel between the server and the clients. Next, the security monitor was enabled, multiple servlet instances were used to serve different clients, and

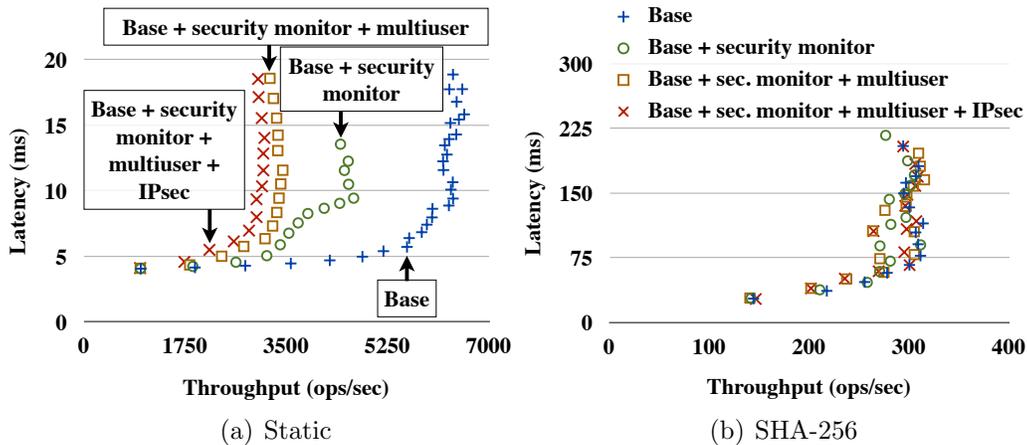


Figure 10.1: Latency vs. Throughput for CleanRoom mechanisms.

IPsec was enabled. For the simple static workload, CleanRoom reduced the throughput of the system by roughly 50%, and incurred an overhead of 0.17 ms per operation. However, for the SHA-256 workload, the computation required to generate the hash effectively hide the overhead of CleanRoom.

To measure the overhead of isolation, the load offered to the server (i.e. the number of requests generated by the clients) and the number of web app containers (i.e. per-client servlet contexts) were varied. Figure 10.2 shows the throughput and response time of CleanRoom for three types of workloads. The static and SHA-256 workloads were the same as in the previous experiment. In the app workload, clients requested a list of documents (about 300) and a specific document (5 to 10 KB) from the servlet. This caused many I/O intensive operation on the small HBase instance that stored the document. As shown in the figure, the overhead of isolation is insignificant for all three

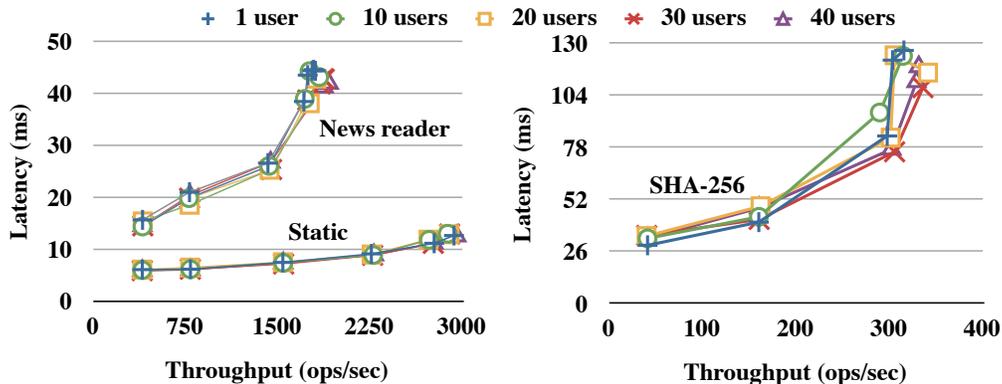
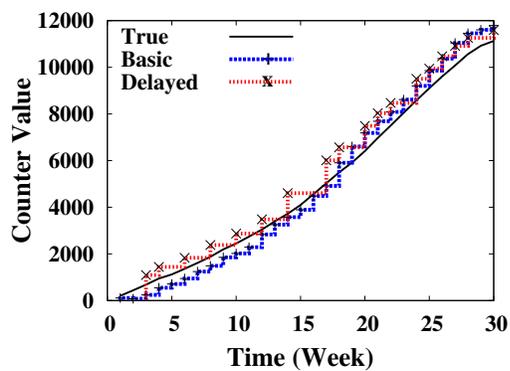


Figure 10.2: Overhead of isolation for various workloads.

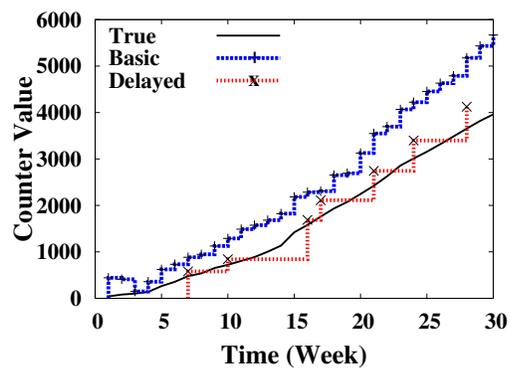
workloads.

## 10.2 Privacy vs accuracy

To illustrate the advantages of the delayed-output mechanism for releasing infrequently updated counters, the traces from University of Saskatchewan web server were used [13] which contained a variety of access patterns. For this experiment,  $\epsilon$  was set to 1, the total number of delayed-output counters ( $|L|$ ) was set to 100, the buffer size ( $b$ ) was set to 500, and the release frequency was set to 1 week. Figure 10.3 shows the values of the delayed-output counter and the basic counter, which simply outputted its differentially private value every week, over a 30-week span for two documents with different access patterns. For the frequently accessed document, the delayed-output counter was off by 12.9% on average vs 19.6% for the basic counter. For less frequently accessed document, the delayed-output counter was much more accurate, with a relative error of 15.6% vs 83.1% for the basic counter.



(a) Frequently accessed



(b) Infrequently accessed

Figure 10.3: Accuracy of delayed-output counter on two different documents when  $\epsilon = 1$ ,  $|L| = 100$ , and  $b = 500$ .

# Chapter 11

## Related Work

*This chapter briefly describes some of the previous works that are done in area of preserving users privacy*

xBook [52] and the system of Viswanath et al. [54] employ an extended sandbox mechanism similar to CleanRoom for social-networking services. These systems protect user information stored on the platform (e.g. users' profiles and social relationships). Unlike CleanRoom, none of these systems can protect private information that apps directly receive or infer from their interactions with the users. xBook anonymizes app statistics (with no information privacy guarantees), while the system of Vishwanath et al. uses conventional differential privacy. As described in chapter 5, this can lead to high relative errors when releasing rarely updated values.

Bubbles [53] aims to capture privacy intentions in a context-centric way, by clustering data into *bubbles* based on explicit user behavior. The privacy guarantees are similar to CleanRoom. However, Bubbles is limited to apps that run on the client device only, whereas CleanRoom supports apps that communicate with a cloud back-end.

Embassies [36] is somewhat similar to CleanRoom in that it aims to

secure apps through minimal interface that allows most apps to function correctly. Unlike CleanRoom, apps publishers are not viewed as adversaries with respect to the user data collected by the app.

Dynamic taint analysis tracks the flow of sensitive data through program binaries [24, 34, 57] and can help protect user privacy. For example, TaintDroid [31] detects (rather than prevents) privacy violations, while AppFence [35] uses data shadowing and exfiltration blocking to prevent tainted data from leaving the device. Neither system handles implicit leaks. While taint-based systems can track specific data items such as device ID, they cannot prevent the app from leaking information about the user’s behavior (e.g., articles the user has read). In general, dynamic taint tracking is complementary to the guarantees provided by CleanRoom. For example, it can be used to prevent certain data items from being declassified even via differentially private channels.

Preserver [37] aims to achieve better privacy by giving control over user’s data to the user. Instead of exposing raw data, it encapsulates user’s data and allows the user to specify which code can access her data and how data can be accessed. This approach is effective only for applications that do not require raw access to the user’s data.

PINQ [42] and Airavat [50] are centralized platforms for differentially private computations on static datasets. PDDP [23] is a distributed differential privacy system in which participants maintain their own data.

While the cloud provider is trusted in CleanRoom, CloudVisor [56] and CryptDB [48] focus on untrusted clouds. CloudVisor hides users' data from the hypervisor using nested virtualization, CryptDB uses encryption. CLAMP [47] employs isolation and authentication mechanisms that are similar to CleanRoom to protect private data in LAMP-like Web servers. It focuses on compromised servers rather than malicious applications.

There have been efforts in providing stronger security guarantees through trustworthy computing environment. This environment assumes the presence of a specialized hardware called *Trusted Platform Module* (TPM) which can be used to generate and store cryptographic keys along with capabilities like remote attestation and sealed storage. Nexus [51] is one such effort which aims to provide secure execution environment to applications through new OS design which better utilize TPM. Terra [33] on the other hand uses virtualization on top of TPM to provide secure environment. Both these efforts provide strong security guarantees for applications running on PCs, however, it is yet to be seen if these techniques can be used on mobile devices. First, mobile devices are limited in their compute power so performance is a concern. Second, these devices often run many applications so scalability will be critical. Moreover, these application use remote cloud for their operation and so any technique which provides security need to take remote cloud into consideration.

CleanRoom can be viewed as imposing a mandatory information flow policy on untrusted apps. Previous work on information flow control includes [25, 39, 55] [46]. The classical IFC system [38] and modern mandatory

access control system require an administrator to statically allocate labels and specify declassification policies. Decentralized information flow control (DIFC) allows applications to allocate their labels and declassify the labeled data [46]. Language based DIFC [11, 45] systems extend the type system by labeling variables with security attributes, which are then enforced by the compiler. OS-based DIFC systems, such as Asbestos [30], HiStar [55], and Flume [39], provide DIFC properties at OS abstractions. It allows applications to express security policies by labeling OS resources such as processes, files, and sockets.

CleanRoom enforces information flow at a much higher level of abstraction (discussed in chapter 6) as compared to language or process level. By doing this CleanRoom simplified its IFC model, but at the same time lost the ability to control flow of information at a much finer-level (e.g. language or process level)

Recently  $\pi$ Box [41] has introduced the app platform design that protects user privacy from untrusted apps.  $\pi$ Box's key idea is to extend the sandbox from mobile device to the cloud and provide platform controlled storage and communication channel. While  $\pi$ Box showed that many useful consumer apps can be built in its environment, it is difficult to build apps that support extensive sharing among its users and thus is not suitable for enterprise apps where collaboration among employees is one of the prime functionality expected from an app.

Bring-Your-Own-Device approaches that support dual workspaces [16, 20] enable personal and corporate data to coexist on the same device while

permitting only trusted apps to access the corporate data. CleanRoom takes this idea a step further and allows untrusted apps to run on corporate data, thus realizing the idea of Bring-Your-Own-App.

## Chapter 12

### Future Work

*This chapter discusses some aspects of BYOA that are not supported in the current design of CleanRoom and how they can be incorporated in future*

**Debugging.** One of the major challenges with CleanRoom is supporting debugging. Since app components run in sandboxes and cannot communicate with the app publisher, the publisher cannot obtain detailed bug reports. One potential solution is to use symbolic execution (e.g., [22]) to generate new inputs to the program, that can reproduce the same bugs and can be submitted in place of actual user inputs. Challenges of using this approach include applying symbolic execution to CleanRoom’s distributed setting and handling more complicated types of inputs (e.g., touch, gesture, pictures). Another possible solution is to use the aggregate channel. For example, the aggregate channel can be used to count how often various exceptions are thrown without sacrificing privacy. The limitations of this approach includes the ability to provide only limited information on the errors (e.g., the type of exception caught) and the limited number of times that the app can report (due to limitations on differential privacy and the privacy budget; see [41]).

**Covert channels.** The existence of covert channels is a challenge for all platforms, including CleanRoom. One covert channel that exists in CleanRoom involves a viewer component running in user sandbox spawning a particular editor component in a data sandbox depending on its knowledge of the user’s private data. This covert channel has highly limited bandwidth: the maximum leakage from each instantiation of the data sandbox component is  $\log_2(N)$ , where  $N$  is the number of blobs this user has access to. Moreover, CleanRoom requires the user to confirm every time a data sandbox component is spawned which implies that any leakage requires the app to exhibit suspicious behavior: either spawning a data sandbox component with a blob different from what the user expected, or creating a set of blobs with very similar names to disguise from the user that a different blob was opened. It remains to be seen whether there exist other channels that may potentially risk the privacy provided by CleanRoom.

**Privacy across blobs.** As stated earlier 4.3, it is possible that when a user chooses to copy some content from one blob to another, a malicious app could smuggle and hide additional confidential content in the latter blob. If a user, unaware of the hidden content, shares the blob, the user may end up sharing the confidential content with users that should not have access to this data. One way to address this issue is to simply prompt users who share data in this fashion and warn them of the possible consequences. Another is to enforce restrictions on who can be granted access to the receiving data sandbox, e.g.,

users that have access to the receiving data sandbox must always remain a subset of the sending data sandbox. A related approach would require that users put blobs into some sort of collection before transferring content between them. Permissions would then be granted on the collection as a whole rather than individual blobs.

**Enterprise-level declassification.** CleanRoom currently restricts the export of data to human-readable formats. This restriction—while safer than allowing arbitrary data—would likely preclude the use of many apps. A feasible alternative is to leverage the availability of resources that enterprise environments have at their disposal: various departments (e.g., IT and legal) that can:

1. write code that examines any exported data,
2. log and audit any exports, and
3. contact publishers directly regarding their apps.

By augmenting existing mechanisms—prompting the user whenever export is about to occur and restricting export to data sandboxes only—with these techniques, CleanRoom could enable many useful capabilities (e.g., emailing arbitrary data to recipients outside the company), while significantly increasing the difficulty of leaking data.

## Chapter 13

### Conclusion

This thesis presented CleanRoom, a new app platform that enables enterprises to use BYOD and BYOA policies without sacrificing the ability to collaborate or the privacy of users and the enterprise. CleanRoom supports rich app functionalities while preserving privacy through: (1) strong isolation spanning both the user device and the cloud, (2) specialized storage and communication channels, and (3) the adaptation of the theoretical framework of differential privacy under continual observation. CleanRoom requires minimal change in the app architecture and so app developers can continue to develop their apps in the same way as they were being developed before. CleanRoom design led to a simplified information flow model with simple rules for sharing and declassification. The evaluation of CleanRoom demonstrates that CleanRoom incurs a very modest performance overhead and can support a variety of applications.

## Appendix

# Appendix 1

## Contributions

CleanRoom is a research project being worked upon by a team comprising of Sangmin Lee, Deepak Goel, Edmund Wong, Mike Dahlin and Vitaly Shmatikov at The University of Texas at Austin. CleanRoom shares its core architecture with its parent project  $\pi$ Box [41]. Sangmin Lee has contributed a great deal towards this project. He designed and implemented the sandboxes both on the device side by modifying the Android kernel and on the cloud by using jetty webserver security mechanisms. He also developed the specialised storages - private vault, app storage, and blob storage by providing wrappers around HBase.

My contributions include collaboration on design and implementation of differentially private aggregation channel. I also contributed in design and implementation of blob service that enables sharing functionality in CleanRoom, development of an information flow control model of CleanRoom, implementation of six sample apps and porting of two open source apps to  $\pi$ Box / CleanRoom platform, detailed analysis of apps from Google play that can be supported by  $\pi$ Box, and design guidelines for building more apps on CleanRoom.

Some chapters in this thesis have been adapted from [41], particularly, chapters discussing differential privacy, evaluation and related work. The write-up has been modified to take into account the modifications introduced by CleanRoom. For example, the chapter on differential privacy was modified to describe how CleanRoom provides privacy at blob-level instead of user-level, and related work chapter has been updated with more work than were discussed in  $\pi$ Box.

# Index

*Abstract*, vi  
*Acknowledgments*, v  
*Android*, 42  
*Appendix*, 78  
*Applications*, 52  
  
*Background*, 9  
*Bibliography*, 90  
  
*Conclusion*, 77  
  
*Dedication*, iv  
*Design*, 18  
*Design Guidelines*, 62  
*Differential Privacy*, 27  
  
*Evaluation*, 65  
  
*Future Work*, 74  
  
*Hbase*, 46  
  
*Implementation*, 42  
*Information Flow Model*, 33  
*Information Flow*, 12  
*Introduction*, 1  
  
*Jetty WebServer*, 45  
  
*Privacy*, 9  
  
*Related Work*, 69  
  
*Threat Model*, 14

## Bibliography

- [1] Android. <http://www.android.com/>.
- [2] Cloud report: Popular apps like Dropbox and Skype are banned by some companies, embraced by others. <http://tabtimes.com/feature/deployment-strategy/2012/08/03/cloud-report-popular-apps-dropbox-and-skype-are-banned-some>.
- [3] First iOS malware hits App Store. <http://www.forbes.com/sites/adriankingsleyhughes/2012/07/06/first-ios-malware-hits-app-store>.
- [4] Google app engine. <https://developers.google.com/appengine>.
- [5] Google enables private Play stores. [http://www.theregister.co.uk/2012/12/07/google\\_announces\\_private\\_channels\\_in\\_play\\_for\\_byod\\_apps](http://www.theregister.co.uk/2012/12/07/google_announces_private_channels_in_play_for_byod_apps).
- [6] HBase. <http://hbase.apache.org>.
- [7] IBM bans Dropbox, Siri and rival cloud tech at work. [http://www.theregister.co.uk/2012/05/25/ibm\\_bans\\_dropbox\\_siri](http://www.theregister.co.uk/2012/05/25/ibm_bans_dropbox_siri).
- [8] iOS social apps leak contact data. <http://www.informationweek.com/news/security/privacy/232600490>.

- [9] Is Google helpless to stop the scourge of Android malware? <http://www.digitaltrends.com/mobile/who-can-fight-android-malware-not-google>.
- [10] Jetty. <http://www.eclipse.org/jetty>.
- [11] Jif: Java information flow. <http://www.cs.cornell.edu/jif>.
- [12] NativeClient - native code for web apps. <http://code.google.com/p/nativeclient>.
- [13] Saskatchewan-HTTP - seven months of HTTP logs from the University of Saskatchewan WWW Server. <http://ita.ee.lbl.gov/html/contrib/Sask-HTTP.html>.
- [14] Selinux project wiki. <http://selinuxproject.org/>.
- [15] Twitter apologizes for squirreling away iPhone user data. <http://www.csmonitor.com/Innovation/Horizons/2012/0216/Twitter-apologizes-for-squirreling-away-iPhone-user-data>.
- [16] VMware Horizon Mobile. <https://blogs.vmware.com/euc/2012/08/vmware-horizon-mobile-on-ios.html>.
- [17] 300,000 mobile apps stealing personal data. <http://www.itproportal.com/2010/07/29/300000-mobile-apps-stealing-personal-data>, July 2010.
- [18] Mário S. Alvim, Miguel E. Andrés, Konstantinos Chatzikokolakis, Pierpaolo Degano, and Catuscia Palamidessi. Differential privacy: on the

- trade-off between utility and information leakage. *CoRR*, abs/1103.5188, 2011.
- [19] Torben Amtoft, John Hatcliff, and Edwin Rodríguez. Precise and automated contract-based reasoning for verification and certification of information flow properties of programs with arrays. *ESOP'10*, 2010.
- [20] Jeremy Andrus, Christoffer Dall, Alexander Vant Hof, Oren Laadan, and Jason Nieh. Cells: A virtual mobile smartphone architecture. In *SOSP*, 2011.
- [21] Gilles Barthe and Boris Kopf. Information-theoretic bounds for differentially private mechanisms. In *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium*, CSF '11, pages 191–204, Washington, DC, USA, 2011. IEEE Computer Society.
- [22] Miguel Castro, Manuel Costa, and Jean-Philippe Martin. Better bug reporting with better privacy. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 319–328. ACM Press, March 2008.
- [23] Ruichuan Chen, Alexey Reznichenko, Paul Francis, and Johannes Gehke. Towards statistical queries over distributed private user data. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2012.

- [24] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 22–22, Berkeley, CA, USA, 2004. USENIX Association.
- [25] Dorothy E. Denning. A lattice model of secure information flow. *Communication of the ACM*, 19(5):236–243, May 1976.
- [26] Irit Dinur and Kobbi Nissim. Revealing information while preserving privacy. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '03, pages 202–210, New York, NY, USA, 2003. ACM.
- [27] Cynthia Dwork. Differential privacy. In *ICALP*, pages 1–12. Springer, 2006.
- [28] Cynthia Dwork, Frank Mcsherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *In Proceedings of the 3rd Theory of Cryptography Conference*, pages 265–284. Springer, 2006.
- [29] Cynthia Dwork, Moni Naor, Toniann Pitassi, and Guy N. Rothblum. Differential privacy under continual observation. In *Proceedings of the 42nd ACM symposium on Theory of computing*, STOC '10, pages 715–724, New York, NY, USA, 2010. ACM.

- [30] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *Proceedings of the 20th Symposium on Operating Systems Principles*, Brighton, UK, October 2005.
- [31] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of OSDI 2010*, October 2010.
- [32] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. In *SOUPS '12*.
- [33] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, 2003.
- [34] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. Practical taint-based protection using demand emulation. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 29–41, New York, NY, USA, 2006. ACM.

- [35] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for: retrofitting Android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security, CCS '11*, pages 639–652, New York, NY, USA, 2011. ACM.
- [36] Jon Howell, Bryan Parno, and John R. Douceur. Embassies: Radically refactoring the web. In *NSDI*, 2013.
- [37] Jayanthkumar Kannan, Petros Maniatis, and Byung-Gon Chun. Secure data preservers for web services. In *Proceedings of the 2nd USENIX conference on Web application development, WebApps'11*, pages 3–3, Berkeley, CA, USA, 2011. USENIX Association.
- [38] Paul A. Karger, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn. A retrospective on the vax vmm security kernel. *IEEE Trans. Softw. Eng.*, 17(11):1147–1165, November 1991.
- [39] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans, Kaashoek Eddie, and Kohler Robert Morris. Information flow control for standard os abstractions. In *In SOSp*, 2007.
- [40] Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, October 1973.
- [41] Sangmin Lee, Edmund L. Wong, Deepak Goel, Mike Dahlin, and Vitaly Shmatikov.  $\pi$ Box: A platform for privacy-preserving apps. <http://>

[apps.cs.utexas.edu/tech\\_reports/reports/tr/TR-2101.pdf](http://apps.cs.utexas.edu/tech_reports/reports/tr/TR-2101.pdf).

- [42] Frank McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. *Commun. ACM*, 53(9):89–97, September 2010.
- [43] J. Mick. Android wallpaper app stole scores of users’ data, sent it to China. <http://www.dailytech.com/Android+Wallpaper+App+Stole+Scores+of+Users+Data+Sent+it+to+China/article19200.htm>, July 2010.
- [44] Sara Motiee, Kirstie Hawkey, and Konstantin Beznosov. Do Windows users follow the principle of least privilege? Investigating User Account Control practices. In *SOUPS ’10*.
- [45] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *In Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, 1999.
- [46] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the sixteenth ACM symposium on Operating systems principles, SOSP ’97*, pages 129–142, New York, NY, USA, 1997. ACM.
- [47] Bryan Parno, Jonathan M. McCune, Dan Wendlandt, David G. Andersen, and Adrian Perrig. CLAMP: Practical prevention of large-scale data leaks. In *Proceedings of the 2009 30th IEEE Symposium on Security*

*and Privacy*, SP '09, pages 154–169, Washington, DC, USA, 2009. IEEE Computer Society.

- [48] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 85–100, New York, NY, USA, 2011. ACM.
- [49] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 199–212, New York, NY, USA, 2009. ACM.
- [50] Indrajit Roy, Srinath Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. Airavat: Security and privacy for MapReduce. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2010.
- [51] Alan Shieh, Dan Williams, Emin Gn, Sirer Fred, and B. Schneider. Nexus: A new operating system for trustworthy computing. In *In 20th SOSP Workin-Progress Session*, 2005.
- [52] Kapil Singh, Sumeer Bhola, and Wenke Lee. xBook: redesigning privacy control in social networking platforms. In *Proceedings of the 18th confer-*

- ence on USENIX security symposium*, SSYM'09, pages 249–266, Berkeley, CA, USA, 2009. USENIX Association.
- [53] Mohit Tiwari, Prashanth Mohan, Andrew Osheroﬀ, Hilfi Alkaff, Elaine Shi, Eric Love, Dawn Song, and Krste Asanovi. Context-centric security. In *Proceedings of the 7th USENIX conference on Hot topics in security*, HotSec'12, 2012.
- [54] B. Viswanath, E. Kiciman, and S. Saroiu. Keeping information safe from social networking apps. In *Proceedings of the 2012 ACM workshop on Workshop on online social networks*, pages 49–54. ACM, 2012.
- [55] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 263–278, Berkeley, CA, USA, 2006. USENIX Association.
- [56] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 203–216, New York, NY, USA, 2011. ACM.
- [57] Qing Zhang, John McCullough, Justin Ma, Nabil Schear, Michael Vrable, Amin Vahdat, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Neon: system support for derived data management. In *Proceedings*

*of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, VEE '10, pages 63–74, New York, NY, USA, 2010. ACM.*