The Interdisciplinary Center, Herzlia

Efi Arazi School of Computer Science

Automatic trust based segregation for mobile devices

M.Sc. Dissertation

Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science (M.Sc.) Research Track
in Computer Science

Submitted by Oren Poleg
Under the supervision of
Dr. David Movshovitz (IDC)

June, 2015

Acknowledgments

I would like to express my deepest appreciation to Dr. David Movshovitz. Without his guidance and persistent help this dissertation would not have been possible. His thorough knowledge of information security was a valuable asset for this work. Dr. Movshovitz invested a substantial amount of time, beyond the regular office hours, so I could fulfill my commitments and for this I want to extend my sincere gratitude.
I would also like to thank Prof. Anat Bremler-Bar for her encouragement and assistance.

I feel privileged to have worked together with Dr. Movshovitz and Prof. Bremler-Bar. Working with such professionals was an inspirational experience for me.

Last but not least, *my dear wife Einat who provided me with the time and space to write this paper.*

Abstract

Mobile devices have become an essential part of our lives. These devices hold much of our personal information such as contact lists, calendar appointments and private messages.

Writers of applications for mobile devices are trying to get hold of personal information. Application such as WhatsApp and Viber needs this information for justified causes, as their business is built on sending messages to people in the contact list. Other applications may use personal information for targeted advertisement, and sometimes for malicious purposes.

Our goal in this paper is to create a mechanism that will enable **separation between records of data, based on sensitivity**. Once separated, we can limit the access to sensitive data thus preventing its leakage.

Assume a corporate that wants to provide the employees with a corporate contact list and a corporate calendar. Some applications should have access to the corporate data. For example, the calendar app should provide a unified view of corporate and personal meetings. At the same time, to prevent leakage, the corporate will not want unsanctioned 3rd party applications to access the data.

The security mechanism of the Android OS is insufficient when trying to protect sensitive user information. Firstly - the security is too coarse, it is enforced in the API level and cannot discriminate between data items within a data container (sensitive and non-sensitive contact items for example). Secondly - the mobile operating systems use the user as the gate keeper. They depend on the user to take a difficult security decision – is the application they want to install safe or not. If the user grants the access, the application will gain access to the entire data set (e.g. the entire contact list or the entire calendar), and if not – the application will not be installed.

iOS shares some of the problems. It also enforces security on the API level and requires the user to take security decisions. It does, however, allow the user to install applications and postpone security decisions to when a secure API is first called. iOS also allows the user to change their decision at any time.

To effectively protect private information from malicious smartphone apps, in this paper, we apply **a record level access control model** to the information stored on the device. Our access control model is based on DAC and Capabilities.

We define an owner for each created record, and create a trust relationship between applications. The owner can grant access to sensitive data to other application by dispensing tickets. Applications that can provide a valid ticket may access the sensitive data, while other applications will be ignorant to its existence.
The access control will effectively block the leakage of sensitive data, but will allow untrusted applications to operate on non-sensitive data.

We have implemented these concepts on Android. Our system consists of a basic access control scheme and code that enforces this scheme on the running applications.
Our proof of concept indicates that the access control support requires only minor modifications in the Android framework, i.e. less than 400 lines of code (LOC).

We also have evaluated our access control model with Android apps that are known to leak contact information. Our results show that we can effectively control the access accesses and protect private information.

Table of Contents

Table of Figures

1. Introduction

Until the late 20th century, contact lists were stored on rolodex. If someone wanted to grab your list, they were required to have physical access. Nowadays, you would probably store all of your contact information on your smartphone, which means that your entire contact list and calendar are probably continuously transmitted to various 3rd parties.

Current surveys show that malwares, poorly written applications and even legitimate applications will leak your sensitive information to the internet [1], while current security measures are not effective in protecting the users' sensitive data.

We want to define subsets of the user's data (Contact list, Calendar, SMSs) that will only be available to the creating entity and to entities that have the creators trust. Untrusted 3rd party applications will not be aware to the sensitive data nor could they access the data, thus will not be able to leak it.

For example, our access control model will enable a corporate application to create a corporate contact list which will reside in the same database as the regular contacts of the Android contact application. The corporate will be able to share this list with other applications developed by the corporate, a couple of applications developed for the corporate by a trusted entity and the system contact application but not to other applications.

This problem is the real. Most smart phone users (84%) [4] mix business with personal use, 70% of the users install applications on the device while 50% of the devices have access to the corporate mail. This means that users have both private data and sensitive corporate data on their smartphone. Corporates which allow employees to connect to their enterprise systems via the smartphone and access corporate data, would like to ensure this sensitive corporate data does not leak from the employee smartphone.

The problem is common to all major mobile operating systems (Android, iOS and also windows mobile). Our focus in this paper is on the Android operating system. The Android environment is open source, which allows us to modify it to implement our security models. It is also the most common in the market, according to a study done by Nielsen (2/2013) [2], [3] (Figure 1).

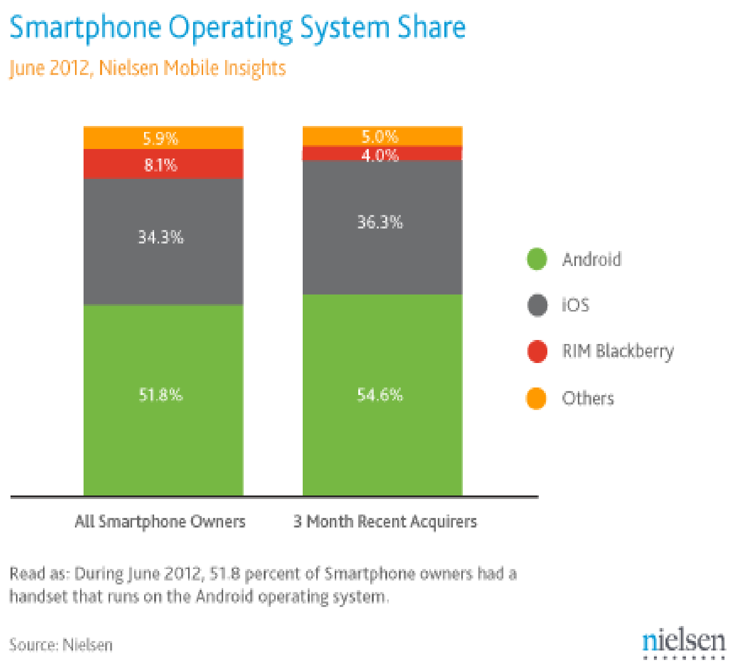


Figure 1: Smartphone Operating System Share

The environment we will be using for our work is a mobile device with Android OS. The user controls the device, and can install 3rd party applications on it. We have no control over the user's response to the security prompts during installation.

We assume that the Android security model is intact:

1. **The underlying OS is trusted** - including the kernel, system services and the android framework.
2. **System applications** (system contact manager / system calendar) **are benign from privacy protection perspective** and will not release private data of the device without authorizations.
3. **No 3rd party application has root privileges**. (Although currently there are some system bugs which allow application to elevate privileges, we assume these will become more scarce with time)

Current protection schemes that are used in the industry (chapter 2) and suggested in prior works (chapter 3) are inefficient in separating sensitive information (such as corporate data) from insensitive one, as they treat the entire data set as a monolithic unit and force the user to act as the security administrator.

In this paper, we argue the need for finer granularity when handling data containers such as the contact list and calendar, to allow control in the record level. We also argue that the user of the phone does not have the capabilities for managing data security, and that the owner of the data should control the permission for the data.

Our security model adds an owner to each data record, and allows this owner to control the access of the record by other applications. 3rd party applications can handle non-sensitive data in the same manner they currently do, while access to sensitive data is controlled by the owner of the data.

In the later chapters we review the various access control models (chapter 4), define an access control model to facilitate our security requirements (chapter 5), and discuss our proof of concept implementation for Android OS, performance tests the results (chapter 6). Finally, we draw the conclusions and discuss extensions to the work (chapter 7).

2. Current Solutions for Information Leakage and their Inadequacy

According to the Cloud Security Alliance Report [1], 2 out of the top 8 security threats to mobile are related to information leakage - Information-Stealing Malware and Poorly written application.

In this chapter we will review the way information is being leaked from mobile device and the current mechanisms that are used to prevent the leakage. We will argue that these mechanisms are too coarse and cannot separate sensitive (e.g. corporate) data items from non-sensitive ones.

2.1. Information Leakage

2.1.1. Information stealing malware

It is fairly easy to grab the entire phonebook of an android device and transmit it to the net (less than 10 lines of code), as seen on Code 1.

```
Cursor phones =
getContentResolver().query(ContactsContract.CommonDataKinds.Phone.CON
TENT_URI, null,null,null, null);
while (phones.moveToNext())  {
String name=
phones.getString(phones.getColumnIndex(ContactsContract.CommonDataKinds.Phone.
DISPLAY_NAME));
String phoneNumber =
phones.getString(phones.getColumnIndex(ContactsContract.CommonDataKinds.Phone.
NUMBER));
        // Send over to your favorite site on the network
}
phones.close();
```

Code 1

The user only needs to accept a permission request dialog. If installed from the play store, which is the preferred installation method, the permission dialog will only state the application uses Contact information (Figure 2: Play

Store Installation). This is very innocent looking and most users, which grew accustomed to the permission dialogs, should have no problems to accept. When installation is done from the file system, more data is displayed to the user (Figure 3: File system installation), but this method is rarely used.

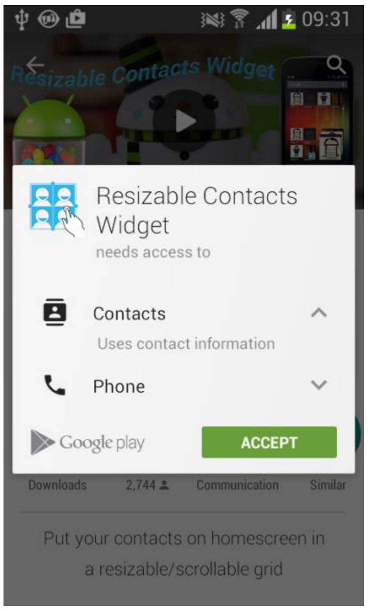

09:31
Resizable Contacts Widget
Resizable Contacts Widget
needs access to
Contacts
Uses contact information
Phone
Google play
ACCEPT
Downloads
2,744
Communication
Similar
Put your contacts on homescreen in a resizable/scrollable grid

Figure 2: Play Store Installation

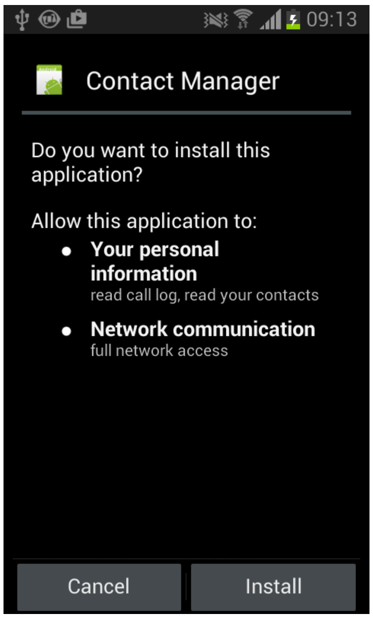

09:13
Contact Manager
Do you want to install this application?
Allow this application to:
Your personal information
read call log, read your contacts
Network communication
full network access
Cancel
Install

Figure 3: File system installation

A behavioral study [15] shows that only 17% of participants paid attention to permissions during installation, and only 3% actually understood the meaning of the various permissions.

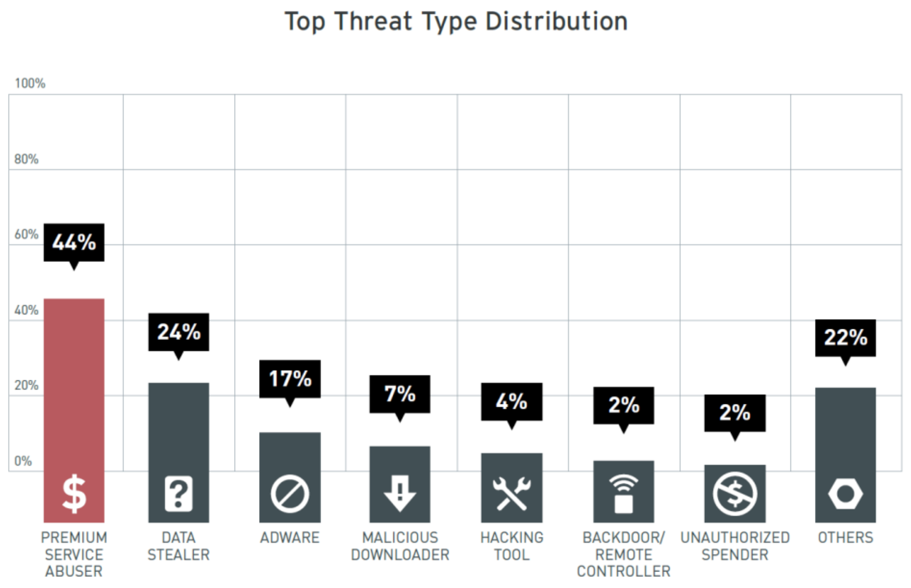


Figure 4: Top Threat Type Distribution

This is not just a theoretical threat. TrendMicro's research found that Data stealing malware account for 24% of malwares [5] (Figure 4).
According to another report there are two reasons for stealing the contact list - they can be used by spammers as part of their distribution list, or sold in lots for prices of .14 - 1.5 Yen per account. [6]

Another source for concern is the alternative app markets. The App Genome Project [7] analyzed two alternative markets for Android that target Chinese users. While these markets serve a legitimate need for localized apps, they also host pirated and repackaged apps. Nearly 11% of the apps also available on the Android Market were found to be repackaged and likely submitted by someone other than original developer. A quarter of the repackaged apps request more permissions than the original app, which is often the effect of a third-party adding an illegitimate ad network or malicious code having functionality such as making premium rate phone calls, sending premium rate SMS messages without the user's knowledge or stealing personal information. There have been multiple instances where repackaged apps in alternative Android markets have served as hosts for malware.

2.1.2. Legitimate application

Some of the applications that need to access the contact list for legitimate reasons will transmit the personal information back to their own servers.

The LinkedIn application, for example, had a privacy issue [8] where calendar information was collected from the device and sent to LinkedIn servers without notifying the users.

Other applications, such as Viber and WhatsApp, are known to transmit the contact list to their servers as part of their normal operation. According to the privacy commissioner of Canada [9], all phone numbers from the mobile device are transmitted to WhatsApp to assist in the identification of other WhatsApp users. Rather than deleting the mobile numbers of non-users, WhatsApp retains those numbers (in a hashed form).

TaintDroid [10] shows that among 30 popular third-party Android apps, there are 68 instances of potential misuse of users' private information. D2Taint [11] reports that over 80% of them leak data to third-party destinations; 14% leak highly sensitive data. Both papers are discussed in the next chapter.

2.2. Android Security [12]

To protect personal information, Android OS incorporated several layers of security. The personal information is stored in database files on the file system. Linux file system permission are used to prevent 3rd party applications from having direct access to the database files, and need to use specially designed Android APIs to access the data. The APIs are implemented using Inter Process Communication, with security checks placed in the API level.

When comparing smartphones to traditional environments, such as multi user UNIX machines computer we observe that the smartphones usually have only one user, and multiple applications. The smartphone applications assume the role of the users in the traditional environments while the end user needs to assume the role of the system administrator. However, most end-users are not capable of managing the system security.

Android Platform is built on top of the Linux kernel. The Kernel provides Android with several key features including a user-based file system permissions, Process isolation, and Extensible mechanism for secure IPC.

The Android platform uses the Linux user-based protection as a means of identifying and isolating application resources. During installation the Android system assigns a unique user ID (UID) to each Android application and runs the application as the assigned user in a separate process.
The kernel enforces security between applications and the system at the process level through standard Linux facilities. By default, applications cannot interact with each other and have limited access to the operating system.

Access Control to Content Providers

In Android, content providers manage access to a structured set of data. They encapsulate the data, and provide mechanisms for defining data security. Content providers are the standard interface that connects data in one process with code running in another process.

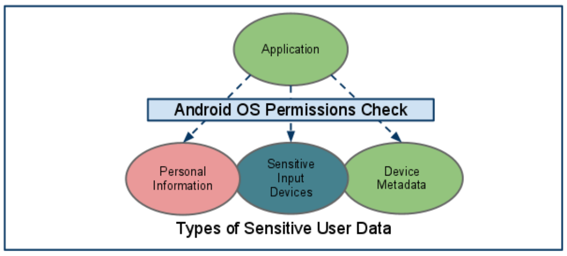


Figure 5: Android OS Permission Check

The android system defines several system controlled content providers, among which are the contact list and the calendar. As applications cannot access the system data directly, they need to use the content provider APIs.

To access the data in the system content providers, an application needs to declare the required access in its manifest, so the system can grant them access permission during installation.

Because system content providers such as contacts and calendar are likely to contain personal or personally identifiable information Android system has defined specific permissions for application that require their usage. Access to the contact list is governed by two permissions: **READ_CONTACTS** and **WRITE_CONTACTS**. Applications having the permission **READ_CONTACTS** can access the entire contact list, while applications having the permission **WRITE_CONTACTS** can modify the contact info. In the same manner, access to calendar data is governed by READ_CALENDAR and WRITE_CALENDER.

During installation, the system will use the declared permissions to prompt the user to approve access to the sensitive data. If permission is granted, the application will be installed and will have access to the data in the relevant content providers.

Users can either grant the application these permissions (as requested by the application), exposing all the contacts data to the application, or not install the application at all. The same applies to other types of information like SMS and emails.

For sharing data between applications, there is a more dynamic option – signature protection level [13]. When this type of protection level is defined for

a component, the system will only grant access if the requesting application is signed by the same certificate as the application that declared the permission (that is – signed by the same developer). The system automatically grants the permission without notifying the user or asking for approval from the user.
This has similarities to our solution, in the sense that no user interaction is required and the dependency of cryptographic certificates.
The differences from our solution are the fact that the Android signature protection is not transferable (the author of the data cannot grant access to other entities, and must be the creator of both applications) and the fact that this protection level relates to the entire data set, and not to the individual data items, which is the level we protect in our work.

2.3. Shortcomings of Android Security segregation

Not enough granularity

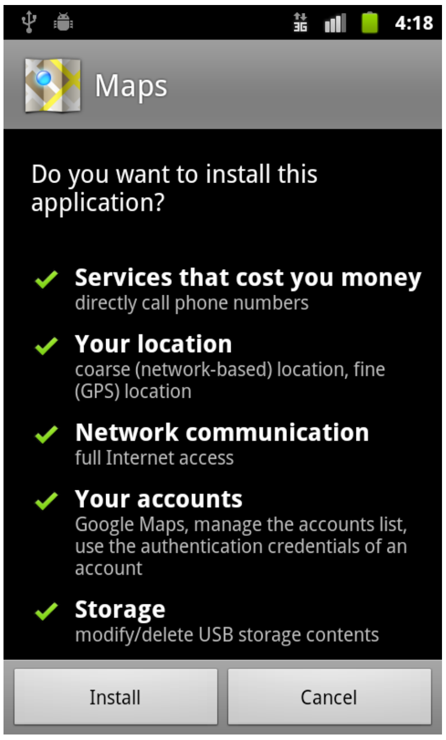
The permissions enforced by the Android system are placed on APIs, cannot be placed only on the sensitive parts of the data. By granting access to a specific content provider, Android will allow access to all of the data handled by it.

The users are faced with the dilemma of usability vs. security. Users want the functionality of the application need to sacrifice all of their personal information.
In study analyzing more than 400,000 applications [14], researchers found that 26% of apps in Google Play require access to personal information such as contacts and email. This means that the users that want to protect their data are limited in the selection of applications they can use.

Users can't manage security

The user needs to agree to the permission request. However, a behavioral study [15] shows that only 17% of participants paid attention to permissions during installation, and **only 3% actually understood the meaning of the**

various permissions. The conclusion is that the user cannot be trusted when it comes to permission management.


4:18

Maps

Do you want to install this application?

✓ **Services that cost you money**
directly call phone numbers

✓ **Your location**
coarse (network-based) location, fine (GPS) location

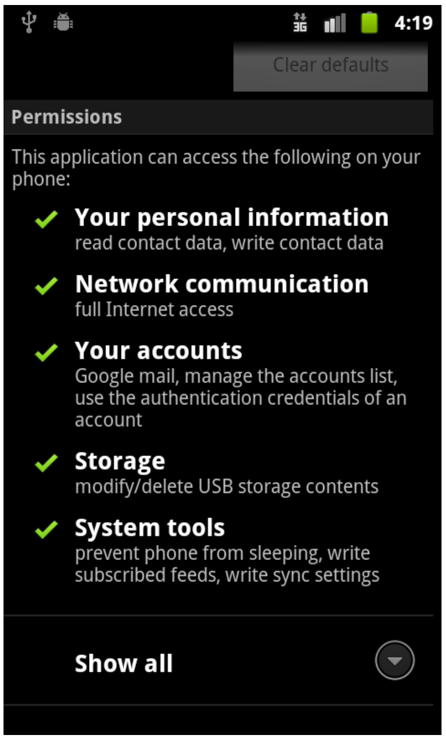
✓ **Network communication**
full Internet access

✓ **Your accounts**
Google Maps, manage the accounts list, use the authentication credentials of an account

✓ **Storage**
modify/delete USB storage contents

Install | Cancel

Figure 6: Typical Android Permission Screen


4:19

Clear defaults

Permissions

This application can access the following on your phone:

✓ **Your personal information**
read contact data, write contact data

✓ **Network communication**
full Internet access

✓ **Your accounts**
Google mail, manage the accounts list, use the authentication credentials of an account

✓ **Storage**
modify/delete USB storage contents

✓ **System tools**
prevent phone from sleeping, write subscribed feeds, write sync settings

Show all

Figure 7: Typical Android Permission Screen

Corporate and users on mobile devices

The boundaries between time at the office and time at home have long been eroded: nowhere is this more evident than in our use of mobile technology. Corporate data will find itself on the employee mobile devices, whether they are issued by the corporate, or are owned by the user ('Bring your own device' - BYOD).

Technically, it is possible to limit download of 3rd party applications, and to restrict the users' activities on the mobile devices, but in practice, employees are becoming less tolerant to such restrictions.

2.4. Industry solutions for Mobile Devices security

Two main models for introducing mobile devices to the corporate are:

- Bring Your Own Device (BYOD),
- Corporate Owned personally-enabled devices (COPE).

The COPE model is easier for IT control of the data as the corporate has tighter control over the applications installed on the device, but according to recent studies [16] loses traction to BYOD.

To resolve the leakage of information, several approaches are currently in use:

Mobile-Device-Management - a policy and configuration management tool for mobile devices.
MDM provides IT with control of the mobile devices, by providing management across four different layers:

- **Software management** - Manages mobile applications, content and operating systems, including provisioning, patches support.
- **Network service management** - Tracks network-device information such as location, usage and cellular/ WiFi to support Provisioning, Billing and Help desk/ support
- **Hardware management** - Manages the physical device components.
- **Security management** - Enforcement of security policies, including: Remote wipe, Remote lock, Secure configuration enforcement, Encryption.

Using MDM the IT can enable users to install only apps that are on a special pre-defined white-list. There are many solutions providers, such as Airwatch, Tangoe, and 3LM, Good. This approach is restrictive from the user point of view and is applicable mostly to the COPE model. [17]

Secure Containers - Solutions which create separation between business and personal data on the mobile to prevent business critical data from leaking out to unauthorized individuals.

This is done by encrypting the data on the phone, encrypting the connection between the device and the enterprise servers and providing additional data security features, such as copy-paste DLP.
Secure containers are complementary to the MDM tools, and provided by as an additional layer. Some examples are Airwatch, FiberLink, Zenprise (Cirtix) and Good Technologies and Samsung Knox.

Remote Desktop solutions such as VMware Horizon View 5.2 and Citrix Receiver.
Corporate applications are executed on the corporate servers or on the cloud, thin client on the mobile device is used to display a desktop from the remove server and send in key events. This usually requires constant network connection, but the corporate data is not stored on the local device.

Light Virtualization solutions such as CELLROX and **General Dynamics Broadband** [18] are providing multiple virtual environments running in parallel on an android device. These solutions create segregation between the virtual instances so that data cannot be moved between the instances, allowing a corporate instance to be separated from the personal instance. These solutions can provide additional segregation by storing the data for corporate instance in an encrypted file, and by tunneling the corporate network over VPN.
ARM is offering support for hardware based virtualization in the Cortex-A15. This will allow tighter security but will limit the number of supporting devices.

While remote desktop and virtualization solutions provide security against data leakage, there are several downsides. The separation between the virtual machines causes the user to handle multiple contact lists and multiple calendars. While searching a contact in a couple of lists may not too cumbersome, working with multiple calendars is a real problem, and may cause the users to try and circumvent the restrictions (e.g. try and duplicate data).

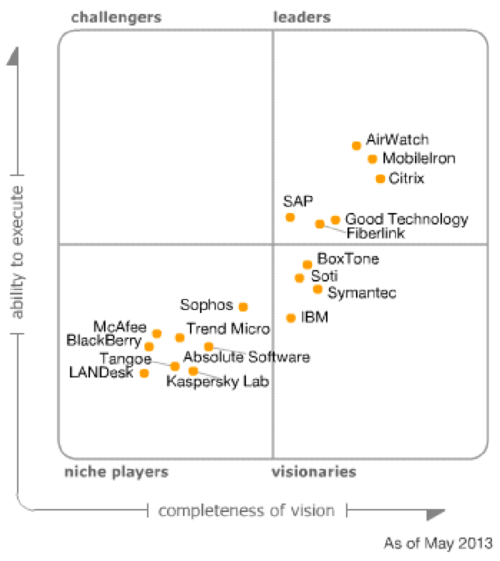


Figure 8: Gartner Magic Quadrant for Mobile Device Management Software

3. Prior works

As the amount of smart phones and tablet devices reaches critical mass, privacy on mobile devices is getting more attention. Multiple papers have been written on the subject of detection and prevention of data leakage from the mobile devices, and several approaches are being pursued.

In our paper, we are describing a solution which allows the owner of the data to control the sensitive data it holds on the mobile device. The mechanism needs to be able to discriminate sensitive data from insensitive one within a content handler and to permit access only to entities trusted by the owner.

We will analyze each solution in this context, and check if the solution a fine enough granularity to separate between types of data that reside in a content handler. We will also check how the permissions are handled – manually or automatically.

MockDroid [19] and **APEX** [20] proposes a lightweight enhancement to Android permission model, allowing users to mock the access from an untrusted app to particular resources at runtime (by reporting either empty or unavailable).

MockDroid focuses on the android package manager for storing the mocked permissions and adds code to the API permission checks used by android as decision points between no permission/ regular permission / mocked data.

APEX creates a flexible policy enhancement for the android. The user can create a policy that will allow an application only a subset of the requested permissions. APEX also allows for dynamic restrictions of policy depending on external variables (e.g. disabling GPS for an application during specific time periods, limiting the number of SMSs an application can send during a time period)

Both solutions allow the user to limit access of applications to APIs and content providers. These approaches apply indiscriminate modifications to the whole data of a content provider, and as such are not granular enough for our purpose. Both solutions need manual control of the user to determine the data flow.

L4Android [21] **and Cells** [22] use virtual machines as means for security around the OS. L4Android proposes a secure microkernel, while L4Android suggests a lightweight operating system virtualization.

Both solutions can provide segregation which is better than the one provided by the android framework. They create multiple segregated instances of Android, each with its own content providers, without a method to get a unified view of the data from all of the content providers. Applications that are installed on the corporate instance will not be able to access the personal data and vice versa.

TISSA [23] is using **security aware content providers** (see diagram below) to control the data flowing into the application.

TISSA uses a UI screen on which the user can set the access level for each application. The levels are - full access, anonymized access (contact names stripped), provide bogus data or return an empty dataset. These preferences are stored in the privacy policy database.

During the retrieval of the data, the content provider queries the privacy setting content provider and returns the appropriate data. (Figure 9)

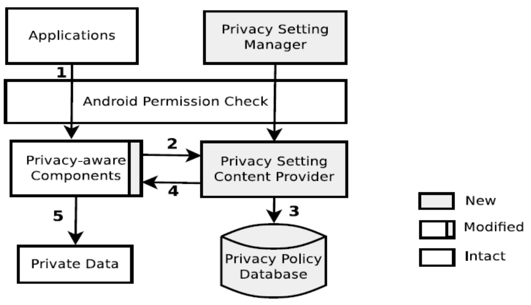


Figure 9: TISSA components

TISSA's approach has similarities to our solution - both are protecting sensitive data with enforcement placed in the Content Provider.

However, there are two major differences in the approaches:
Granularity - TISSA deals with the entire data set in a unified manner, the access levels granted are applied to the entire data set. Our solution is dealing with the data at the record level, and can provide different applications with different subsets of the data, based on the permission level of the application.

User Interaction - TISSA requires manual configuration – the user is required to select the appropriate security level for each application. This task is cumbersome in real world deployment, and requires time and effort from the user, as well as technical knowledge about behaviour of the various applications. Our solution needs no user interaction, and is based on trust the owner of the data has for the relevant applications.

TaintDroid [10] proposes dynamic taint analysis to help prevent data leakage. TaintDroid automatically labels (taints) data from privacy-sensitive sources and transitively applies labels as sensitive data propagates through program system. When tainted data is transmitted over the network, or otherwise leave the system, TaintDroid logs the event. TaintDroid infrastructure is robust, but requires considerable changes to the OS and incurs 14% performance overhead.
Due to the fact TaintDroid have a limit of 32 tag types, tags are assigned only to data origins (e.g. content providers). TaintDroid tags are too coarse for separating sensitive and insensitive data within a content provider.

Follow up works such as **YASSE** [24] and **D2TAINT** [11] extend the TaintDroid infrastructure; improve the tagging mechanism allowing finer control over the transmitted data. While D2TAINT uses this information to notify the user of potential security leak, YASSE have created a security model that allows the user to control the flow of sensitive information on the device.

YASSE supports user defined labels and performing label modifications due to the enforcement of filtering policies on tagged data. The user can tag part of the data (e.g. the sensitive contacts). The tags may be used for differentiating sensitive data within a content handler. However the tagging of the data is manual and the user has to manage the sensitive data. YASSE has created a module for the users to create and manage the rules that control the information flow. During installation the user is prompt for decisions for the new application.

With regards to our approach we can observe that YASSE does enable granular separation of data within a content handler, by tagging it as sensitive.
However, YASSE has no means for creating automatic trust between the data owner and data users. The user controls the data and needs to take complex security decisions and to create the security policy.
Tracking the propagation of the data incurs overhead which is unnecessary if access is managed at the source.

AppFence [25] is working both on the ingress and egress to monitor data leakage. Methods similar to TISSA and Mockdroid are used in the ingress to control the data shared with the applications. AppFence also allows the user to select the records they do not consider sensitive, so they can shared with the applications.
AppFence can allow the sensitive data to enter applications that requires it, but will use an extension to TaintDroid to stop the sensitive data from being transmitted to the network.
AppFence tagging is based on the originating content handler, and cannot discriminate between sensitive and insensitive data that reside in the same content handler. The user needs to create rules to control the flow of the data which was not in the scope of the AppFence paper.

4. Access Control Model

In the traditional access control models described below, the subjects were the users and processes of the system, and the objects were the various system components (files, directories, memory-segments and resources such as printers and processors).

When discussing the mobile devices, there is usually only one user to the system, and the **subjects** we want to control are the applications on the mobile device. Although installed and executed by the user, they may vary in the amount of trust they should receive. The **objects** we wish to protect in this work are the user's private data items stored in the system content providers (e.g. contact list, calendar appointments and emails).

We will start by providing a recap of the traditional models [26] – ACL vs Capabilities and DAC vs MAC, review of the current Access Control Models used in mobile OSs, and follow with a discussion of our suggestion for enhanced Access Control Model.

4.1. ACL vs Capabilities

An access control list (ACL) [27], with respect to a computer system, is a list of permissions attached to an object. An ACL specifies which users or system processes (i.e. subjects) are granted permission access to the object, as well as what operations each subject is allowed to perform on that object. Each entry in a typical ACL specifies a subject and an operation. For instance, if a file has an ACL that contains (Alice, delete), this would give Alice permission to delete the file.
One of the simplest forms of ACL is implemented in all UNIX-like file systems. In the file system, there are 3 classes – **Owner** / **Group** / **Other** and 3 Modes **Read / Write / Execute**. Each file system object (file or directory) is assigned an Owner and a Group from the subjects of the system. For each class (owner, group and other) the file system holds 3 bits that that indicate if read/write/execute are allowed. When a subject wants to invoke an operation

on an object, the system will determine if the subject is the owner of the object, if not – if the subject belongs to the group defined for the object and otherwise it will be treated as other. The system will check the class of the subject, and the access permission required against the bitmask of the object, and will provide file system access accordingly.

In contrast, **Capability-based security** [28] [29] is a concept in the design of secure computing systems. A capability (known in some systems as a ticket) is a communicable, unforgeable token of authority. It refers to a value that references an object along with an associated set of access rights. A user program on a capability-based operating system must use a capability to access an object. Capabilities are typically stored by the operating system in a secure area, with mechanisms to prevent direct manipulation of the capabilities by the users (to prevent forging of access rights or changes to the pointed objects)

Android uses capability-based security (without delegation) in two ways. **Firstly**, each application declares the list of permissions it requires in its manifest using a <uses-permission> element. During installation the installer determines whether or not to grant the requested permission by checking the authorities that signed the applications certificate, and if needed (in most 3rd party applications) prompt the user. The permissions act as capabilities in the sense they are attached to the applications (subjects), and allow the subject to manipulate various aspects of the system, such as the network connectivity and access to the content providers.

Secondly, an application may also protect its own components (e.g. content providers it wants to publish) with permissions. It can employ any of the permissions defined by Android; permissions declared by other applications or define its own. It may also define “signature” protection level, discussed below.

With regards to the permissions used by the Android system, we see the following shortcomings - Android permissions are lacking the dynamic nature of the capabilities, as the capabilities cannot be changed during the lifetime of the application, nor can they be transferred. Furthermore, the permissions only limit the access to APIs and do not provide granularity relating of the data accessed.

With regards to protecting sensitive data, and differentiating it from non-sensitive one, these permissions are not relevant. The permissions only protect the APIs accessing the data, and as such are not able to discriminate between data items within a container. Our solution allows the creator of a data item to protect that item and to control the access to that item.

4.2. MAC vs DAC

Mandatory Access Control (MAC) refers to a type of access control in which the security is controlled by the system, rather than the users. The permissions are not governed by the owner of the resource, nor can they changed by users. Instead, they are set by the security administrator. Subjects and objects each have a set of security attributes. Whenever a subject attempts to access an object, an authorization rule enforced by the operating system examines these security attributes and decides whether the access can take place. Any operation by any subject on any object will be tested against the set of authorization rules to determine if the operation is allowed. The control is Mandatory in the sense that it is controlled by an administrator of the system. The security attributes of the various components are assigned by the administrator, and cannot be changed or assigned by the users (as in DAC). MAC is employed in SE-Linux, which is integrated into Android, and discussed below.

Discretionary access control (DAC) is a type of access control defined by the Trusted Computer System Evaluation Criteria [30] as “a means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject (unless restrained by mandatory access control)".

In Contrast to Mandatory Access Control, DAC will allow the subjects of the system to control access to data which they own, rather than have a global policy that controls access to all information. File system on Linux (and Android) are a simple implementation of DAC (with ACL) in the sense that the owner of the file can manage the ACL and grant permission access to the data to other users of the system, and receivers of a permission can share it with others.

Occasionally a system as a whole is said to have "discretionary" or "purely discretionary" access control as a way of saying that the system lacks mandatory access control. On the other hand, systems can be said to implement both MAC and DAC simultaneously, where DAC refers to one category of access controls that subjects can transfer among each other, and MAC refers to a second category of access controls that imposes constraints upon the first. Linux systems which employ SE Linux extensions are a good example. The traditional UNIX file security is based on DAC (see section 3.1), SE Linux adds another layer on top of DAC, which enforces MAC. While users can control and share the access to the files they create, the system administrator maintains the ability to limit the sharing with system enforced policies.

4.3. Current Access Control Model on mobile OSs

Traditional Android employs two layers of security, application permission model, and kernel level sandboxing and isolation. Each level has its own Access Control Model.

On the application level, Mandatory Access Control is mandated using a permission list, which is enforced by the android middleware. This list controls the applications access to system resources such as network, camera and GPS, and to system content providers that contain sensitive data, such as the contact list and events (It is also used to control IPC between applications' components) .

This model is used by the application developers, who need to specify the required permissions of the application, and to declare them in the application manifest.

During installation, the user will be prompted with the list of permissions requested by the application. Google have identified this as a problem, and are trying to simplify the permission display [31]. The user will be requested to allow these permissions during installation to proceed with the installation process. Failing to allow the permissions will result in installation failure [12] [32].

Once the application is installed it will be granted all of the permissions declared in its manifest. The operating system will store the list of permissions for the installed application, and the applications cannot change the stored permissions directly. This aligns with a capability based system (without delegation of capabilities).

Below the application level, Linux kernel is used for sandboxing and isolation of processes. Android relies on Linux discretionary access control for the isolation in two ways.

First, DAC is used to separate between applications in the same manner users are isolated on conventional Linux systems. Each application is allocated with a unique user identifier and group identifier during the installation. The process of the application is executed with these identifiers, and so is the data (files) created by the application. Each application will create files with permissions modes that only allow access to the owner (the application). This will prevent other applications from accessing data files through file system APIs.

The Linux kernel will effectively prevent other applications from accessing the application data through the kernel interfaces.

Second, DAC is used to separate applications from system resources.

This is done by executing the system services in with system UIDs. Applications which run with their own UID are not able to directly access the system resources. The applications can access the system services using the IPC which has application level permission enforced.

There are specific permissions that leverage the UNIX file system permission models directly [33]. When an application is granted INTERNET, WRITE_EXTERNAL_STORAGE or BLUETOOTH permissions, it will be assigned to a file system group that has permissions to create the relevant sockets or files. The kernel will take charge of enforcing the access control for these permissions, and the API library will operate directly on these descriptors, removing the need for Application level permission checking.

4.4. SE Linux

SE Android [34], SE Linux [35] port for Android, is a collection of modules and kernel patches that allow Mandatory Access Control model to be applied to application layer objects and operations. SE Android features are being migrated into Android Open Source Project (Since release 4.3)

Based on a flexible mandatory access control architecture called Flask [36], its goal is to separate the enforcement of security decisions from the security policy itself. The libselinux library provides interfaces for use by applications to obtain security contexts for their own objects and to apply SELinux permission checks on operations performed on these objects. SELinux uses policy files to store the Mandatory Access Policies. These files are loaded during system startup. The policy files contain rules that use the security contexts allow or deny activities on system resources.
The SELinux permission check happens after the Linux DAC check, so it cannot be used to relax the limits of the usual DAC of Linux system, only to further restrict it.

In Android, SE Linux is being integrated to enhance both the DAC used in the underlying Linux support and to the middleware, by introducing “Install time MAC” control over the installation process, allowing a mandatory policy to be imposed on the installation process, on top of the security decisions taken by the user. A policy (mac_permissions.xml) is used to configure Install-time MMAC policy and provides mapping from developer certificates to SELinux permissions so that the android virtual machine will spawn an app in the correct domain. This means that applications will be executed in specific contexts according to the system policy, enforcing system limits to the permissions that can be assigned by the user.

SELinux improves the security of Android, but does not cover the type of security we are aiming for in our work. It does not deal with the data items granularity we have (it is dealing with files and processes in the Linux level, and not with the data). Also, it imposes Mandatory Access Control, which requires some overall knowledge of the system and applications. This is not appropriate for our goal, which is enabling the sharing of data between applications which are not known in advance.

4.5. Access Control in iOS

With respect to Access Control Models, iOS has a slightly different means of controlling the applications. On the system level, the applications are sandboxed on the file system. According to iOS documentation, each application is assigned its own space on the file system, and has no access to the data of other apps [37].

In the middleware level, iOS employs a dynamic access control rather than static install time control in Android (this feature was introduced in iOS 6). The application does not need to declare the permissions it uses on installation time. When the applications tries to access the sensitive information for the first time (e.g. location data, contact list, events, access to images), the user is prompted to approve or deny access for the application iOS will store the users decisions, and allow the user to change their decisions from the Privacy menu in the Settings(Figure 10). Each permission has a separate menu containing the list of applications that requested the permission, and the current user decision (allow/deny), which the user can modify (Figure 11).

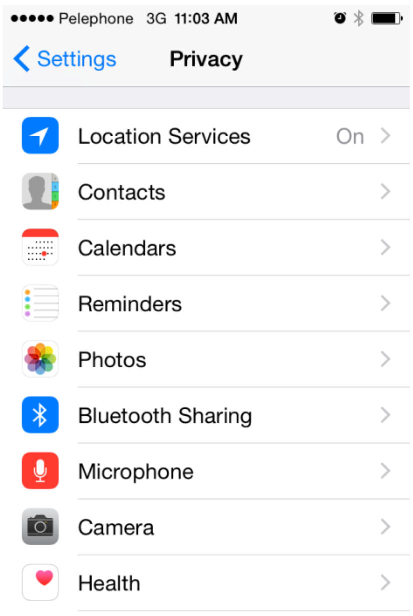
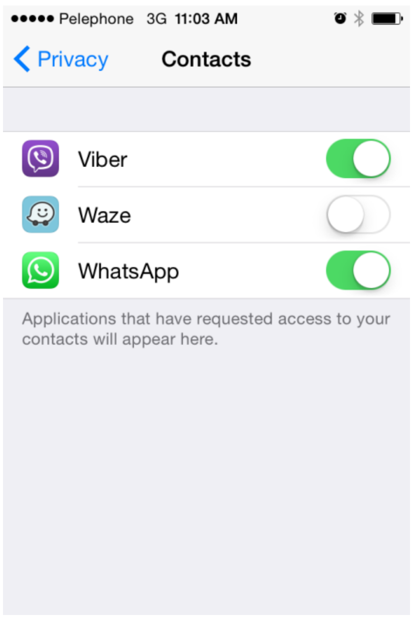


Figure 10: iPhone Privacy Menu



Viber: On
Waze: Off
WhatsApp: On

Applications that have requested access to your contacts will appear here.

Figure 11: iPhone contact submenu

Permissions are placed on specific APIs in the same manner as in Android, and as a result see the entire data set as a monolithic unit. Applications that are granted access to the contact list can see the access the entire contact list (iOS has no differentiation between read and write permissions to the contact list).

It is easy to see that iOS suffers from the same issues we aim to solve in our work: It does not provide means to differentiate between types of data items in the same container (sensitive vs. non-sensitive contacts for example), or to allow only specific applications to access the sensitive data, it also requires the user to control the security of the data manually, rather than establish trust between applications we advocate in our work.

4.6. Conclusions

We have found two fundamental problems with the Access Control Models used on current mobile devices with regards to access of sensitive information.
The first issue is the user's involvement in the security decisions. Android's Install time MAC policy is defined so that only applications that were granted access privileges by the user can be installed. The user becomes the policy

decision point, and is faced with the difficult security questions which he is not always equipped to answer. The user needs to decide, based on very little information, if the application should be granted all of the security permissions it requires, possibly risking the system, or not, losing the applications functionality.
This is a very difficult decision, even for security conscious users. It may be simple to deny access to a flashlight application requiring access to the contact list and GPS, but even a legitimate calendar application that requires Calendar access and Internet access may still leak your calendar appointments to the network, if not written properly.
Furthermore the Access Control is enforced in the API level. The granularity of the permissions is such that Android's contact list only has two permissions – 'read' and 'write' and iOSs contact list access has only one permission. If the application was approved read access to the contact list, it will be able to access the entire contact list, regardless of the sensitivity / source of the data. This means that the entire data collection handled by a content manager needs to be regarded in a unified manner from security perspective, and there is no mechanism to secure parts of it. WhatsApp, for example, is becoming one of the most prominent applications for messaging, and many users require it for conduct their daily business. It will use the Contact List Read permission along with Internet access, allowing it to read (and send to WhatsApp server) the entire contact list. If the user has his sensitive corporate contact list on his device, this too will be sent, and there is currently no way to limit the access of the application to only 'non-sensitive' parts.

Both of the access control models deployed in Android are centralized. Install time permission use the user as the PDP (as in the granting permissions while installing applications), while the SE Android assume knowledge of the users (applications) in the system to create the security policies.
Centralized approach is not adequate when we protect the information at the single data item level. There are too many data items for the user to make decisions for, and there may be too many options for the user to select from.
The users are not capable of deciding at this level of granularity.

The users of the system (applications) may not be known when the data is created. So having a predefined policy in place is lacking for the data which is *dynamic.*

We need a more dynamic access model to deal with the dynamic data.

5. Our Solution

5.1. Overview

In this paper we describe an access control model that can prevent sensitive information from being leaked by untrusted entities, without the need for the user to manage data security.

We separate the **owner** of the data from the **user** of the phone, and allow the **owner** of the data to control the access to the data. **Data** items that can be protected are contact records, calendar entries or any other data item that is controlled by the Android's content provider (iPhone security has similar characteristics, but is outside of the scope of this paper). The **owner** can grant access to the data he owns to other **entities** (applications) in the system by providing them with signed **tickets.** The details of the ticket structure are in section 5.3.

Our system is responsible for verifying the **tickets** and for managing access to the sensitive data according to the information contained in the ticket.

During the creation of a data record a security tag identifying the owner of the data can be attached to the record if it is considered sensitive.
Operations on the data consist of sending a request through IPC, and receiving a result, which may be a list of data items (in query) or count of affected items (in delete). During the operation the data, the security tag will be compared with the ticket provided by the requesting entity. If the requestor is entitled to act on the data according to the credentials (e.g. a corporate application requesting to access a corporate contact), the data item will be included in the results, otherwise, it will be silently omitted, making the caller unaware of the sensitive data.

5.2. Our Access Control Model

To solve the issues discussed previously – creating the security in the data record level, and securing the data without user interaction, we are defining a security model that will be enforced in the data record level, which is a combination of **DAC** with **Capabilities**.

Each data may have an owner. To access data item that has an owner, a valid ticket needs to be presented. The ticked is provided by the owner to the trusted parties to be included in the trusted application. When an access is done to an 'owned' data item, the system will verify the validity of the ticket, and match it to the application that requested the data. If the request is valid, the data will be returned, otherwise, it will be silently ignored.
Capabilities model assumes knowledge of all of the objects. In our model we assume that the owner provides access to all the records he creates. We can refine our model further to support only a subset of the records by extending the entitlements and the enforcement for more granular access control based on data values, or by adding a tag to each created item, and using this tag in the capabilities list. However, due to the fact that an owner may have several certificates/identities, it is also possible to create the subsets by using different identities.

This means we will use **Discretionary Access Control** rather than MAC or RBAC.

Each data record can have an owner, which is assigned during the record creation. The owner can be any entity which has a private/public key pair (similar to Certificate Authorities in X.509). The owner can be the creator of the data, but it is also possible that any 3rd party application creates the data, and attaches the owner's public key to the data items.
The users of the data are entities which are known to the Android system, that is – installed applications. The owner of the data is responsible for dispensing signed tickets to the users of the data (application developers), to

be incorporated in their applications. Application Developers can send their public key to the data owner and receive signed tickets.

During the creation of the data record, the hash of the owner's public key will be attached to the data record. The creator of the data can select the owner of the data or to leave the data without an owner, allowing unrestricted access.

Due to the amount of objects in the system (each record is an object) in comparison to the subjects (number of applications that wish to access the data), and the complex relations that may occur, it is easier both in space and processing time, to attach only a static security tag (owner) to each record, and to determine access permissions by comparing this tag to the ticket the provided by the subject. In this manner, there is no added complexity in install time (e.g. going over all of the objects during application installation and updating an ACL) and minimal processing is only needed during access of the data. As a result we will be using Capabilities.

5.3. Tickets and Entitlements

The **ticket** will contain the identifier of the calling application, and **entitlements** signed with the **private key of the owner** of the record.
The TICKET will be provided along with a TICKET_SIG, which is the digital signature on the ticket data.

TICKET="<Signer package name> <Caller Fingerprint> <Entitlements>"
TICKET – Text strings separated by spaces

- **Signer –** package name of the signing entity (used for quicker lookup in the Package manager)
- **Fingerprint** – SHA-1 hash of the public key of the entity using the ticket (can be modified to use SHA-2)
- **Entitlements –** List of keys and values i.e. Expiry, Allowed actions.

TICKET_SIG="SIGNATURE"

SIGNATURE is a base64 encoding of SHA1WithRSA signature with the private key of the owner on the TICKET string.

We are using plain text tickets for readability. Ticket data can be encoded into X.509 certificates and minimal code changes are needed to use them instead of plaintext.
The X.509 certificates will need to be base64 encoded to be used in the SQL queries.

Entitlements are implemented as a list of keys and values, to provide flexibility in implementation. Basic entitlements are mapped to the basic functionality provided by the content handler - **query**, **insert**, **delete**, **update**, controlling the functionality that the system will allow the user to perform on the data. The **insert** entitlement allows other applications to create records on the **owner's** behalf.
The **update** entitlement allows the calling application to change data in the record (but not the security tag)

To maintain compatibility with standard Android, the following entitlements need to be hard coded:

- Record Owner have all entitlements
- System applications have query/delete/update entitlement. This can be changed in the future if needed.

A known problem of capabilities model is the fact it is not possible to revoke a ticket after it was dispensed. A solution is that the owner may want to issue tickets with short lifespan.

5.4. Ticket Validation

We have taken the original logic of the data operation and moved it to a private internal function. We have created a security aware wrapper around it which the IPC will call instead of the original function.

This method can be applied for each data operation we want to protect (insert/delete/update)

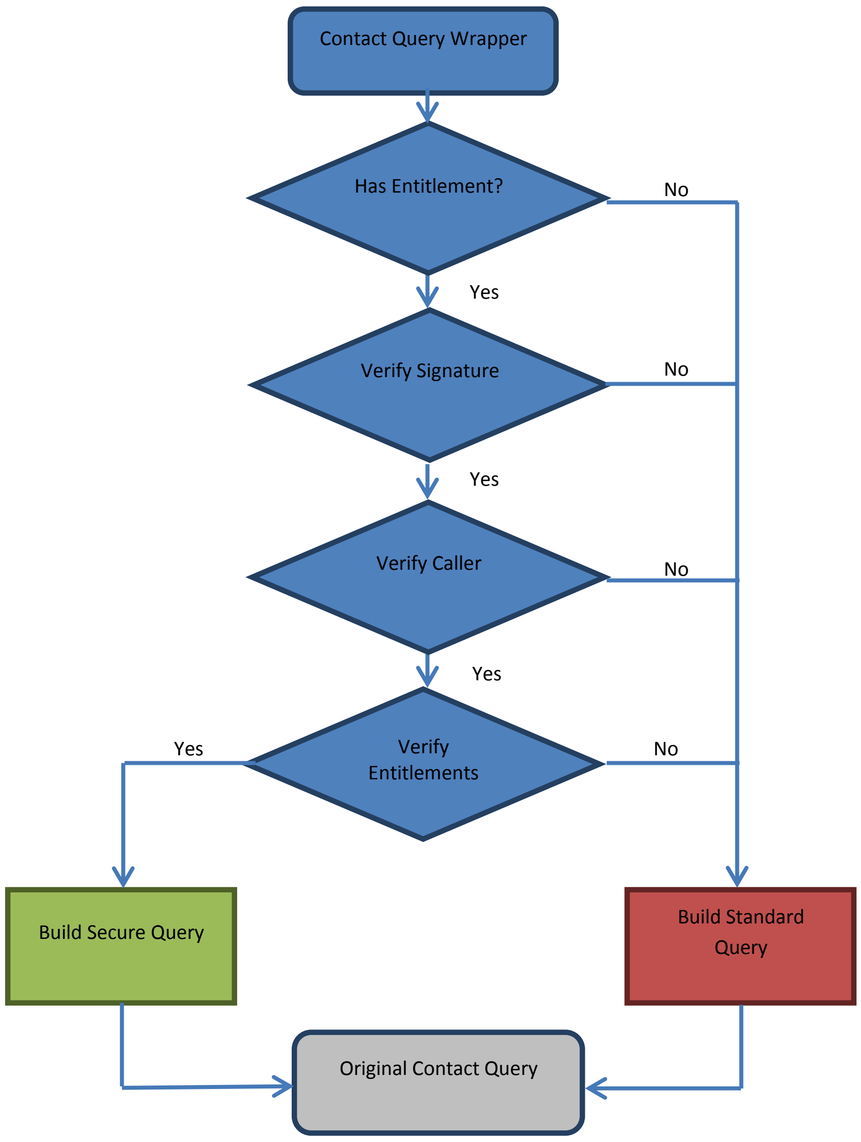


Figure 12: Security Wrapper Verification Flow

When a data operation is executed on the Contact Provider, security wrapper will conduct the following stages (Figure 12):

1. If SQL query contains a ticket entity (AND ticket='data' AND signature='sig'), retrieve the ticket and signature from the SQL query (if not exists, query for standard data, see step 6)
2. Extract the signer of the ticket from the ticket, fetch public key of the signer from Android's package manager.
3. Verify Validity of the ticket (signature) – verify the signature provided with the ticket against the ticket and the public key of the signer (using SHA1RSA)
4. Fetch identity of the caller using Android's package manager, compare to the ticket. (Android holds all public keys for installed applications in the package manager and allows applications to query it. This way we transform the local unique id of the application on the device to a globally unique public key of the application)
5. Verify that the entitlements match the requested operation (A basic string matching of the entitlement name against the current operation)
6. If all checks are ok, builds the secure SQL query based on the data acquired query data with the owner equals to the signer of the ticket, as well as standard data. Otherwise – query only standard data (data without an owner). This is done by manipulating the SQL query text and adding a clause limiting the value of security tag field.
7. Call the original query code with the updated SQL query

6. Implementation

To demonstrate our work, we developed a proof of concept project in which we added our access control model to the Android operating system to create item level security in the Contact Provider. This chapter will review the changes done in the Android operating system, the tests we conducted and the results.

The project includes changes to Android source code, as well as test applications written for Android environment. The changes to the Android source consisted of creating a security wrapper around the Query function of the Contact Provider, limiting access to sensitive contacts. The same methods can be applied to remove and update functions, and to the other content providers.

In our implementation, we have added a field named **ac_security** to the contact content provider database schema. This field will hold the identity of the owner of the field in the insert query, to denote a sensitive data item. Our security wrapper will use this field for operations on sensitive information, as described below.

There are no changes to the content provider APIs (Android's **ContentProviderOperation** class), but only to the underlying implementation. This means that current Android application executables will continue to function with our framework.

Insertion of secure elements requires the creator of the data to add the **ac_security field** to the insertion query and set the field to the id of the owner. Fetching secure elements require the caller to attach a **Ticket** and **Signature** to the SQL query, as detailed in the following sections.

6.1. Design Requirements

When we came to design our access control for Android, we have used the following guidelines to make the implementation viable in real world scenarios: The implementation must have minimal impact on system performance and resources, with little to no noticeable overhead. The implementation must also be compatible with current applications and application security mechanisms. This will also allow us to test our implementation against real applications.

These guidelines imply a lightweight solution that uses Android security mechanisms and builds upon them, with minimal changes to the Android OS.

6.2. Integration with Android's security

Android OS is already separating the sensitive data from 3rd party applications. The OS stores the data we wish to protect (Contact and Calendar events) as database files on the android file system. 3rd party applications have no direct access to the database files. The Android system has defined **content provider interfaces** which encapsulate these databases, and use SQL queries to manipulate them. Each content provider is executed with permissions that allow access to the relevant database. The content providers communicate with other processes in the system via IPC, which is marshaled by the applications' permissions.

6.3. Policy Enforcement

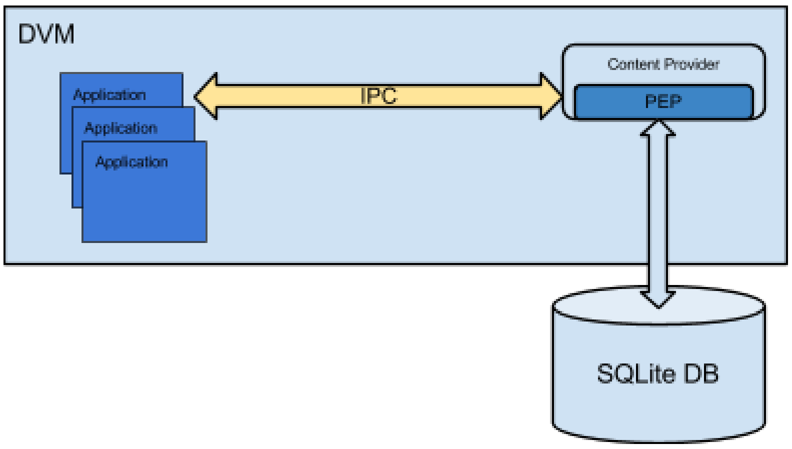


Figure 13: Policy Enforcement in Content Provider

We are implementing the PEP (Policy Enforcement Point) in the content provider. The PEP will run in system context and monitor the IPC requests from the applications. The content provider generates an SQL query based on the IPC request. Our code will add security constraints to the generated SQL before querying the database so that sensitive elements will be only retrieved for callers that have permissions. Insensitive elements will be returned to all callers allowing compatibility with legacy applications.

To support the PEP, we add a field to that will hold the owner information of the data (**ac_security**) to the database we wish to protect (e.g. Contacts database). Each **data record** in the content handler **may** have **ac_security tag** specifying an **owner**. (The **ac_security** tag is derived from the public key of the **Application Developer**, and is discussed in the next section)
Our system can directly verify if the owner is trying to access its own data by matching the accessing application id with the tag of the record. This is done by querying Android package provider for the public key of the calling application and matching it with the **ac_security** tag of the records.

When a different application tries to access the data, it will need to present a ticket to be validated by the system. Our system will be responsible for enforcing the security definitions, using the public keys of the applications that are stored in the Android system.

The **owner of the data** (the holder of the private key which corresponds to the ac_security tag of the data), is responsible for dispensing **tickets** for the records.

6.4. Selecting the record security token

To create a simple AC we will attach a security token to each record, identifying the owner of the record (empty security token if the record is public). This token should uniquely identify the creating application, in a manner that will not allow forging by other apps.

The alternatives we have researched are:

- **Application id** - Each application is assigned a different id during installation. Although unique in the system, this may change when uninstalling/reinstalling application and change between devices.
- **Application name** - Should be unique in the system, but it is easy to create an alternative application with the same name, (removing the original app, installing the new app, getting access to the data)
- **Application public key –** Android employs a public/private key scheme to enable signing of applications.
    - Each developer has a private key, which is used to sign the application during the build process. Android system records the public keys for each installed application during installation.
    - Android's Package Manager can be queried for the public key of a known application.
    - The system holds the mapping between process id and the application information so we can query the system and receive the public key for a specific process.

We will use the public key of the application for generating the security token. This will insure the identification of the data owners and requesters. To reduce the amount of data stored, we will use a digest function on the public key.

6.5. Changes to the schema

The changes to the schema are minimal, and contain only the addition of the security tag, that will be used to indicate a private item. All other changes are reflections of that change in the database views.

In **Raw_contacts** table, which contains the basic contact element, we have added the text field **ac_security**. This field will contain the security token which will be consulted during the fetching of the item.

Added references to the ac_security in the following views: Views.RAW_CONTACTS, Views.RAW_ENTITIES, Views.DATA.

Added reference to **ac_securtity** in the projections used by contact maps.

6.6. Data Creation

Security aware application will create a secure record by attaching a security tag to the record when creating it. A key named "ac_security" needs to be added to the record using standard Android APIs (ContentProviderOperation.Builder.withValue).
The value for "ac_secutiry" is derived from the application's public key. The key is obtained by using Android's package manager APIs, and digest is done using java's security package.
Application can create a record on behalf of other entities providing their key when creating the record. The key can be obtained from the package manager which supports querying information for any installed application.

If a record if created without "ac_security" tag, the record will be accessible to all requesters, enabling compatibility with the legacy applications. Records created by application that are not aware of our security mechanism will be created without the "ac_security" tag.

6.7. Data retrieval

To access sensitive data, an application that uses our framework needs to provide two additional conditions in the SQL selection - TICKET and TICKET_SIG specified in section 5.3.

The following code (Code 2) demonstrates how to attach TICKET and TICKET_SIG information to the selection string parameters in Android's ContentResolver.query().

```
static String selection = "TICKET = 'com.corporate.creator
D2:9E:19:7A:FE:78:CB:E1:5B:E5:2D:17:74:1E:1B:EE RQWU 12/7/2020' AND
TICKET_SIG =
'MoOEWvo/hWh1BW/sEBNN869b2dmfoYLmZxXfpwmkg7iQJgkJ6jZCvnZs4elZSMMH6cc
NIn2aB8SA55/Dv8isUeBw9veKhF8Ou4Sp77ING/lKB5dCUHA/+7Z1dniG0PncdDw/xJ6
uwpjzJIyUSzl3Ht29Ua5Z+H3YpIY1l3l3Ldo='";

static String[] selectionArgs = null;
static String selection = null;

// Run query
private Cursor getContacts()
    {
        return getContentResolver().query(uri, projection,
selection, selectionArgs, null);
    }
```

Code 2: Query with Ticket

The wrapper is responsible for validating the ticket, as described in section 5.4.

If there is no ticket, or ticket is not valid, only records without a security tag or records with a security tag matching the caller will be returned.
If the ticket is valid, the records with the owner matching the dispenser of the ticket will also be returned in addition to the tickets without a security tag and the tickets with security tag matching the caller.

Other operations (update/delete) are not implemented in this POC. The implementation will be done in a similar manner to data retrieval. The caller will provide a ticket which will be verified. Action on sensitive data will only be permitted if the caller is the owner of the data, or if the caller provides a valid ticket from the owner.

In a production system, tools will be used to generate the tickets. The requesting application developer will send his public key to the data owner. The data owner will decide on the entitlements he wishes to provide to the application developer, and will provide a ticket with those entitlements, signed by his own private key. This ticket will be inserted to the application by the developer, to be used in the data queries.

In this POC we crafted the tickets by hand.

6.8. Effectiveness

In our proof of concept project, we created 3 applications for testing the creation and retrieval of contact list data. The first application created the contact list records, and attached a security tag. The second application retrieves the contact list items, with an entitlement signed by the creator. The third tried to retrieve the same contacts, but without the entitlement, to simulate a 3rd party application. We could see that while the owner of the data, and our application with entitlement could retrieve the sensitive data, the application without the entitlement could only retrieve the non-sensitive items. We tested our solution with several 3rd party applications that are known to read (and use) contact information – WhatsApp, Viber, Skype and Instagram.

To demonstrate the effectiveness of our solution in a real world scenario, we installed WhatsApp on an Android emulator. We created 2 contacts – “Regular User” was created by the standard contact manager, and “Most

Hidden Contact" was created by a Security-Aware application, and was tagged as secure.

As seen below, the Contact manager (Figure 14) and the trusted application (Figure 16) can view both contacts, while WhatsApp and other 3rd party applications (Figure 15) can view only the "Regular User".

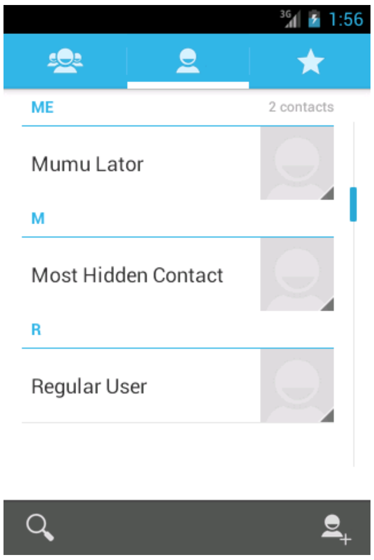

ME 2 contacts
Mumu Lator
M
Most Hidden Contact
R
Regular User

Figure 14: System Application

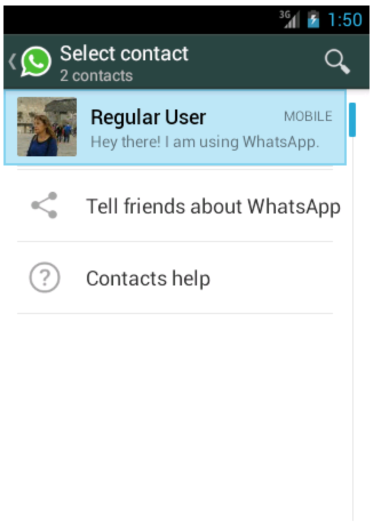

Select contact
2 contacts
Regular User MOBILE
Hey there! I am using WhatsApp.
Tell friends about WhatsApp
Contacts help

Figure 15: Untrusted 3rd party

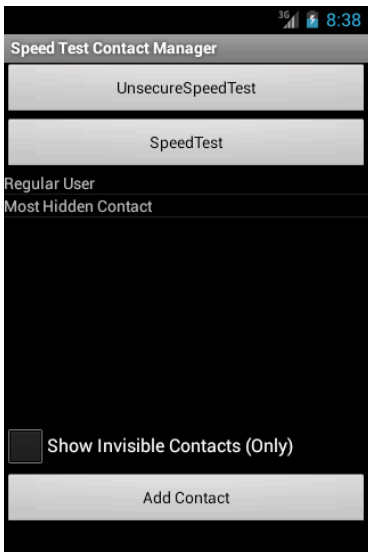

Speed Test Contact Manager
UnsecureSpeedTest
SpeedTest
Regular User
Most Hidden Contact
Show Invisible Contacts (Only)
Add Contact

Figure 16: Trusted App

From the above demonstration we can see that our solution can effectively prevent 3rd party applications from accessing secure information, while allowing access to trusted application and system applications.

6.9. Performance

For performance evaluation, we used executed a test application in 5 execution flows, as described below. Each flow was executed with 4 databases - 500,1000,1500,2000 contacts.

The secure records were created with a standalone application (DataCreator), which has a different public key from the test application. We used the public key of DataCreator to sign the tickets used in the test application.

The test application runs a query to retrieve all contacts from contact provider. During the execution, the database read is repeated 100 times, and the average time (ms) is recorded. Each test was repeated 3 times.

As a reference for time measurement, we used the original Android OS code, without our modifications. Data is created without a security tag. This was recorded as the **baseline (1).**

The rest of the flows are executed against the Android OS that has our modifications.

To evaluate the time difference for regular applications (not aware of our security mechanism), we executed the test application on **data that does not have a security tag**, and sent **no ticket** in the request **(2)**.

To evaluate the performance of regular application with secure data we created a flow where the data is created **with security tag**, but test application does **not provide ticket** to read the data. (No data is returned to the application) **(3)**.

To test normal operation of security aware application, we created a flow in which the **data was created with security tag**, and **our test application provided a ticket** to match the **(4)**.

The final flow simulates an execution of a system application, which has access to all data, and is recognized by application id and not with a ticket. We use data that is created with security tag, and modified the system to treat our test application as **system application (5)**.

# Items		Not Secure	Secure			
		(1) Base line	(2) No Ticket Open Data	(3) No Ticket Secure Data	(4) Ticket And Secure Data	(5) System App
500	1	33.02	41.19	35.08	61.36	32.9
	2	33.44	41.92	36.13	63.71	33.7
	3	33.18	40.34	35.25	60.45	33.1
	Avg	33.21	41.15	35.48	61.84	33.5
1000	1	45.94	59.26	45.11	77.8	46.6
	2	45.02	57.08	46.75	78.07	46.4
	3	46.46	57.79	46.45	79.47	47.7
	Avg	45.80	58.04	46.10	78.44	46.2
1500	1	60.34	73.24	51.66	98.21	59.3
	2	60.58	73.85	52.27	98.23	58.3
	3	61.45	74.84	54.47	98.36	60.5
	Avg	60.79	73.97	52.80	98.26	59.3
2000	1	78.99	96.60	64.88	123.71	83.7
	2	78.11	94.34	64.12	120.7	78.8
	3	78.79	99.10	63.83	123.23	76.5
	Avg	78.63	96.68	64.27	122.54	79.0

Table 1: Performance Test Results

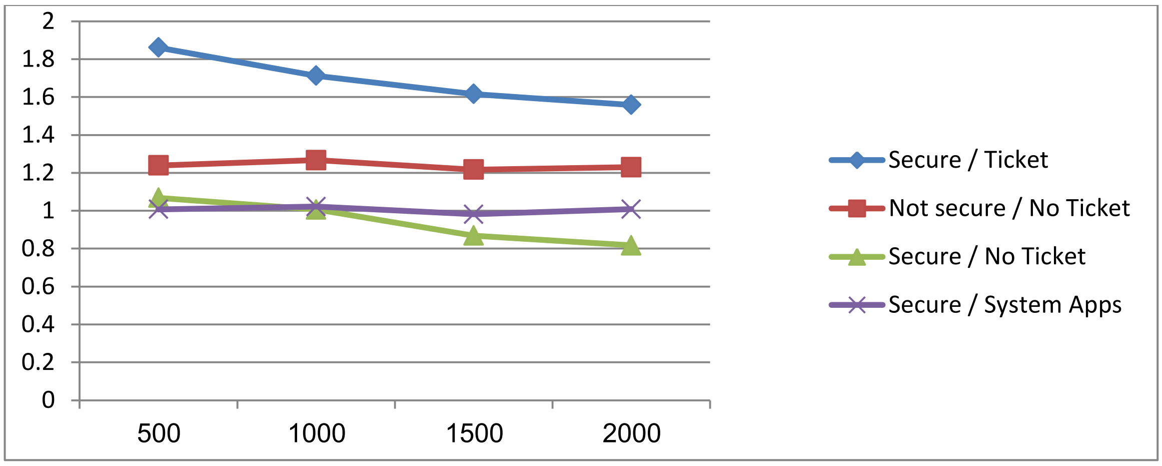


Figure 17: Ratio compared to base flow

In our performance test, in the worst case, our solution takes 2 times the running time of the non-secure version. The ratio declines as the # of data item increases (Figure 17).

All the time differences measured are bound by 60ms. According to Nielsen's Usability Engineering [38], 0.1 second is about the time limit for having the user feel that the system is reacting instantaneously. Thus our work does not have any perceivable effect on the user.

In actual applications, such queries are executed sporadically and not in a loop, as in our tests, so the time difference is not accumulated.

7. Summary and Conclusions

In this paper, we argued for the need for security in mobile smartphones which should be data centric, rather than API based.
The need comes from the fact that smartphone applications will intentionally leak users' private information to their backend servers.

As a solution, we designed an ACL based security on top of the current Android security mechanisms. Our solution allows the sharing of sensitive information between trusted applications while withholding this information from the untrusted ones.
Our approach is Orthogonal to current Android permissions, and is transparent to unaware applications.

Our POC shows that our solution is able to provide separation between contacts based on tags embedded in the data. System applications and trusted applications are able to access the data while untrusted 3rd party applications are ignorant to its existence.

We also show that the performance impact of our solution is not noticeable to the user. (43 ms extra for iterating 2000 contact items), making our solution viable from the user experience aspect.
Our experiments demonstrate its effectiveness and practicality. The performance measurements show that our system has a low performance overhead.

In our POC we've protected the query functionality of the contact provider. Similar protection can be deployed on delete, and update. This work can also be extended to other content providers, such as calendar. Additional work can be done in allowing the ACL to work with multiple owners of data.

We have focused our work on Android OS. However, the same security issues apply to other mobile operating systems, such as iOS and Windows mobile. Our preliminary study shows that the same methods can be applied to iOS, but due to the closed nature of the iOS environment, this work needs to be done in cooperation with Apple.

Our work relies on the integrity of the underlying Android OS. There are virtualization solutions that provide creation of several instances (or personalities) of the Android OS on the same device, each containing its own data. We have discussed the shortcoming of these solutions as they create fragmentation of the user data between the instances (section 2.4). It should be possible to integrate the contact and calendar data of all the virtual instances into one, and use our work to provide each instance with the data it should access. (Provide corporate sensitive data only to the corporate instance and personal information only to the personal instance, while providing non-sensitive data to all instances)

Our work can also be extended to improve the implementation of the capabilities model. It is possible to extend the trust relationship in a manner that the holders of a ticket can delegate capabilities to entities they trust, by creating a chain of trust.

Record level protection we suggest in our work has implementations for database protection. We did not find any literature discussing a distributed database that does not have a central security authority. We believe that our work can be extended to provide a mechanism for distributed databases. The security of the records will be controlled controlled by the data owners and the database will enforces the policy in the same manner our current work implemented it for android.

References

[1] Cloud Security Alliance, "Security Guidance for Critical Areas of Mobile Computing, V1.0," [Online]. Available: https://downloads.cloudsecurityalliance.org/initiatives/mobile/Mobile_Guidance_v1.pdf. [Accessed 2013].

[2] Coalfire, "BYOD Survey: 47 Percent of Users Lack a Password on Smartphones Accessing Company Files," [Online]. Available: http://www.businesswire.com/news/home/20120814005332/en/BYOD-Survey-47-Percent-Users-Lack-Password. [Accessed 2013].

[3] Nielsen, "Nielsen:Two thirds of new mobile buyers now opting for smartphones," [Online]. Available: http://www.nielsen.com/us/en/newswire/2012/two-thirds-of-new-mobile-buyers-now-opting-for-smartphones.html.

[4] Nielsen, "Nielsen Report 2013," [Online]. Available: http://www.nielsen.com/content/dam/corporate/us/en/reports-downloads/2013%20Reports/Mobile-Consumer-Report-2013.pdf.

[5] E. H. S. E. A. H. E. C. D. W. Adrienne Porter Felt, "Android Permissions: User Attention, Comprehension, and Behavior," in *Symposium on Usable Privacy and Security (SOUPS)* , 2012.

[6] Trend Micro, "TrendLabs 2Q 2013 Security Roundup," [Online]. Available: http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/reports/rpt-2q-2013-trendlabs-security-roundup.pdf. [Accessed Jan 2014].

[7] Trend Micro, "Digging Deeper Into ANDROIDOS_CONTACTS.E's Data Stealing Routines," [Online]. Available: http://blog.trendmicro.com/trendlabs-security-intelligence/digging-deeper-into-androidos_contacts-es-data-stealing-routines/.

[8] Lookout Mobile Security, "The App Genome Project," [Online]. Available: https://www.lookout.com/resources/reports/appgenome#chapter6.

[9] A. S. a. Y. Amit, "LinkedOut - A LinkedIn Privacy Issue," [Online]. Available: http://blog.skycure.com/2012/06/linkedout-linkedin-privacy-issue.html.

[10] Office of the privacy commissioner of canada, "WhatsApp's violation of privacy law partly resolved after investigation by data protection authorities," [Online]. Available: http://www.priv.gc.ca/media/nr-c/2013/nr-c_130128_e.asp.

[11] P. G. B.-G. C. L. P. C. J. J. William Enck, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *9th USENIX Symposium*, 2010.

[12] X. L. G. L. A. C. C. Z. C. F. Q. a. D. X. Boxuan Gu, "D2Taint: Differentiated and Dynamic Information Flow Tracking on Smartphones for Numerous Data Sources".

[13] Google, "Android Security Overview," [Online]. Available: http://source.android.com/devices/tech/security/index.html.

[14] Android, "Android Permissions," [Online]. Available: http://developer.android.com/guide/topics/manifest/permission-element.html.

[15] Bit9, "Pausing Google Play: More Than 100,000 Android Apps May Pose Security Risks," October 2012. [Online]. Available: https://www.bit9.com/files/1/Pausing-Google-Play-October2012.pdf.

[16] C. A. |. MSPmentor, "COPE Receives Big Blow from Gartner, BYOD Gains Momentum,"

[Online]. Available: http://mspmentor.net/mobile-device-management/cope-receives-big-blow-gartner-byod-gains-momentum.

[17] L. Seltzer, "Wanted: Secure Android App Whitelisting," [Online]. Available: http://www.informationweek.com/mobile/mobile-devices/wanted-secure-android-app-whitelisting/d/d-id/1103022?.

[18] "http://www.ok-labs.com/," [Online].

[19] A. R. N. S. S. Alastair R. Beresford, "MockDroid: trading privacy for application functionality on smartphones".

[20] S. K. X. Z. Mohammad Nauman, "Apex: Extending Android Permission Model and Enforcement with User-Defined Runtime Constraints," in *5th ACM Symposium on Information, Computer and Communications Security*, 2010.

[21] S. L. A. L. A. W. a. M. P. M. Lange, "L4android: a generic operating system framework for secure smartphones," in *1st ACM workshop on Security and Privacy in Smartphones and Mobile Devices*, 2011.

[22] C. D. A. V. H. O. L. a. J. N. J. Andrus, "Cells: a virtual mobile smartphone architecture," in *SOSP*, 2011.

[23] X. Z. X. J. V. W. F. Yajin Zhou, "Taming Information-Stealing Smartphone Applications (on Android)," in *20th Network and Distributed System Security Symposium*, 2013.

[24] B. C. E. F. Y. Z. Giovanni Russello, "YAASE: Yet Another Android Security Extension".

[25] S. H. J. J. S. S. a. D. W. P. Hornyack, "These aren't the droids you're looking for: retrofitting android to protect data from imperious applications," in *CCS*, 2011.

[26] N. W. Group, "RFC4949 - Internet Security Glossary, Version 2," [Online]. Available: http://tools.ietf.org/html/rfc4949.

[27] Wikipedia, "Access Control List," [Online]. Available: http://en.wikipedia.org/wiki/Access_control_list.

[28] H. M. Levy, "Capability-Based Computer Systems," 1984. [Online]. Available: http://homes.cs.washington.edu/~levy/capabook/. [Accessed 2014].

[29] Wikipedia, "Capability based security," [Online]. Available: http://en.wikipedia.org/wiki/Capability-based_security.

[30] D. O. D. STANDARD, "TRUSTED COMPUTER SYSTEM EVALUATION CRITERIA," December 1985. [Online]. Available: http://csrc.nist.gov/publications/history/dod85.pdf.

[31] Google, "Review app permissions," [Online]. Available: https://support.google.com/googleplay/answer/6014972?hl=en. [Accessed September 2014].

[32] M. O. P. M. William Enck, "Understanding Android Security," [Online]. Available: http://css.csail.mit.edu/6.858/2013/readings/android.pdf.

[33] E. C. S. H. D. S. D. W. Adrienne Porter Felt, "Android Permissions Demystified," [Online]. Available: http://www.cs.berkeley.edu/~emc/papers/android_permissions.pdf.

[34] S. S. a. R. Craig, "Security Enhanced (SE) Android: Bringing Flexible MAC to Android," [Online]. Available: http://www.cs.columbia.edu/~lierranli/coms6998-7Spring2014/papers/SEAndroid-NDSS2013.pdf. [Accessed June 2014].

[35] National Security Agency, "Security-Enhanced Linux," 2009 Jan 15. [Online]. Available: http://www.nsa.gov/research/selinux/index.shtml. [Accessed Jun 2014].

[36] N. S. A. Stephen Smalley, "Flask: Flux Advanced Security Kernel," 26 Dec 2000. [Online]. Available: http://www.cs.utah.edu/flux/fluke/html/flask.html. [Accessed Jun 2014].

[37] Apple , "The iOS Environment," 23 10 2013. [Online]. Available:

https://developer.apple.com/library/ios/documentation/iphone/conceptual/iphoneosprogrammingguide/TheiOSEnvironment/TheiOSEnvironment.html. [Accessed Jun 2014].

[38] J. Nielsen, Usability Engineering, 1993.

[39] C. K. E. K. G. V. Manuel Egele, "PiOS: Detecting Privacy Leaks in iOS Applications," in *18th Annual Network and Distributed System*, 2011.

[40] P. H. K. A. a. H. B. Georgios Portokalidis, "Paranoid android: Zero-day protection for smartphones using the cloud," in *Proceedings of the 26th Annual Computer Security Applications Conference*, 2010.

[41] Nielsen, "Nielsen: America’s New Mobile Majority: a Look at Smartphone Owners in the U.S.," [Online]. Available: http://blog.nielsen.com/nielsenwire/online_mobile/who-owns-smartphones-in-the-us/.

תקציר

התקנים ניידים (טלפונים סלולריים, מחשבי טאבלט) הפכו לחלק בלתי נפרד מחיינו. התקנים אלו מכילים מידע האישי שלנו כגון אנשי הקשר, יומן פגישות והודעות טקסט.

כותבי יישומים להתקנים הניידים מנסים להשיג גישה למידע האישי הזה. לעיתים הגישה נדרשת מוצדקת. אפליקציות כגון WhatsApp ו-Viber משתמשות במידע כדי להקל על המשתמש לצור קשר עם אנשי הקשר שלו. לעיתים האפליקציות מעוניינות במידע לצרכי פרסום ממוקד, ולעיתים למטרות זדוניות.

מנגנוני אבטחת המידע במערכות ההפעלה הניידות (Android/iOS) לא מהווים הגנה מספקת מאפליקציות זדוניות. הם דורשים מהמשתמש להחליט בזמן התקנת הישום – האם היישום המותקן בטוח או זדוני. אפליקציה שהמשתמש אישר תקבל גישה לכל אנשי הקשר (או פגישות היומן) ואין אפשרות לבחירה יותר עדינה של תת קבוצה של המידע.

כדי לייצר הגנה אפקטיבית של המידע האישי מאפליקציות זדוניות, אנחנו מציעים ליישם מודל של בקרת גישה על המידע המאוכסן בהתקן. אנחנו מגדירים בעלים לכל רשומה ויחס של אמון בין יישומים. הבעלים של מידע יכול להעניק גישה למידע ליישום בו הוא בוטח. המודל ימנע דלף של מידע רגיש, בזמן שיישומים שאיננו בוטחים בהם יוכלו לפעול על מידע לא רגיש.

מימשנו את התפיסות הללו על מערכת Android. המערכת שלנו מורכבת מסכמת בקרת גישה פשוטה, וקוד שאוכף אותה על היישומים המותקנים.
הוכחת ההיתכנות שלנו מראה שהתמיכה בבקרת הגישה דורשת שינויי תשתית ב Android, אבל שינויים אלה קטנים מאוד (פחות מ400 שורות קוד)

בדקנו את המערכת מול יישומי Android שמנסים לשלוח את רשימת הכתובות לשרתים ברשת האינטרנט. מהתוצאות ניתן לראות שאנחנו יכולים לחסום גישה למידע הפרטי ברשימת הכתובות ובו בזמן לאפשר גישה ליתר המידע.

מרכז הבינתחומי בהרצליה
בית-ספר אפי ארזי למדעי המחשב

בידול מידע אוטומטי מבוסס אמון במערכות ניידות

מוגש כחיבור סופי לפרויקט מחקר לתואר מוסמך

על-יד אורן פולג

העבודה בוצעה בהנחיית ד"ר דוד מובשביץ

יוני, 2015