# Enforcing a fine-grained network policy in Android

W.P. (Mark) van Cuijk

August 2011

# Enforcing a fine-grained network policy in Android

Master Thesis

by

W.P. (Mark) van Cuijk

August 23, 2011

Supervisors:
Dr. ir. L.A.M. (Berry) Schoenmakers
Ir. M.B. (Maarten) Nieuwenhuis
Ir. M.B. (Matthieu) Pâques, CISA, CISSP

Eindhoven University of Technology
Department of Mathematics and Computer Science

**Abstract**

To prevent malicious applications from causing harm to the user of a device, the Android platform has a permission model. Applications are required to possess permissions when they want to perform operations that may incur monetary cost or violate confidentiality and integrity of personal data stored on the device. One of the resources that has been protected by a permission is access to the Internet. In practice, 60% of the applications that are available in the Android market request this permission, giving them access to communicate with any host on the Internet.

This thesis presents the permission model used by Android and the way it has been implemented. It also discusses several vulnerabilities that have been identified in existing research literature, including several proposals for enhancing the model. The contribution of this thesis is the proposal of an enhancement to the permission model that allows a more fine-grained Internet access policy to be enforced, implementing the principle of least privilege to Internet access by applications on the Android platform. A proof of concept has been created to demonstrate the feasibility of the enhancement.

# Acknowledgements

I would like to show my gratitude to my supervisor Berry Schoenmakers, especially for helping me finding a suitable topic for this thesis and for giving constructive feedback at times I was not sure about the directions to follow.

This thesis would not have been possible without the help and support of my colleagues at KPMG. In particular, I want to thank Maarten Nieuwenhuis and Matthieu Pâques for the time they have devoted to my project. Among other things, it has inspired me that, regardless of other work, you both took time during holidays, weekends and evening hours and even scheduled review blocks with high priority in your calendars.

Other colleagues at KPMG I want to thank in special are Peter Kornelisse, Erwin Hansen and Marjoleine Kruijf. You all have helped me a lot with getting started at KPMG. I also want to thank William Breuer for proofreading the entire thesis.

Special thanks go out towards other Kerckhoffs Institute students graduating at KPMG: Pieter Rogaar, Brinio Hond and Martijn Sprengers. In particular, I want to thank Pieter for helping me to better understand the paper about SCanDroid.

My education and this thesis would not have been possible without the support of my parents and I want to thank them for their support over the years. In particular, I want to thank my mother for being helpful with putting things in perspective when making important decisions.

Lastly, I offer my regards and blessings to all of those who supported me in any respect during the completion of the project.

Mark van Cuijk

# Table of Contents

**Acknowledgements** **iii**

**Table of Contents** **v**

**List of Figures** **vii**

**1 Introduction** **1**
1.1 Research question . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.2 Subquestions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.3 Thesis overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
1.4 Related work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

**2 The Android permission model** **9**
2.1 Android architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.2 Applications . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.3 Android security model . . . . . . . . . . . . . . . . . . . . . . . . . . 12
2.4 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

**3 Implementation details of the permission model** **19**
3.1 Inter-component communication . . . . . . . . . . . . . . . . . . . . . 19
3.2 Enforcing permissions . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
3.3 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

**4 Known vulnerabilities of the permission model** **25**
4.1 Permission-free operations . . . . . . . . . . . . . . . . . . . . . . . . 25
4.2 Permission-related shortcomings . . . . . . . . . . . . . . . . . . . . . 27
4.3 Executing applications . . . . . . . . . . . . . . . . . . . . . . . . . . 31
4.4 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

**5 Existing improvements to the permission model** **33**
5.1 Changing permission granularity . . . . . . . . . . . . . . . . . . . . . 33
5.2 Kirin security policy invariants . . . . . . . . . . . . . . . . . . . . . . 34

5.3 Information flow analysis with SCanDroid . . . . . . . . . . . . . . . . 36
5.4 Apex permission model extensions . . . . . . . . . . . . . . . . . . . . 41
5.5 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
5.6 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

**6 Improving the permission model: the introduction of fine-grained Internet permissions 47**
6.1 Constraining sockets . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
6.2 Comparison of the potential solutions . . . . . . . . . . . . . . . . . . 50
6.3 Socket state machine . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
6.4 Implementation details . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
6.5 Socket policy . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
6.6 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67

**7 Conclusion and future work 69**
7.1 Proof of concept . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
7.2 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
7.3 Future work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72

**Bibliography 75**

**Glossary 79**

**Index 81**

# List of Figures


1.1 DroidWall configuration activity . . . . . 4
1.2 Firewall iP prompting the user for network access . . . . . 5
1.3 BlackBerry prompting the user for network access . . . . . 6

2.1 System architecture of the Android platform . . . . . 9
2.2 Sandboxing and component interaction . . . . . 13
2.3 Permissions screens for an application . . . . . 15
2.4 Extension of content permissions . . . . . 17

3.1 Inter-process communication between Java objects . . . . . 20
3.2 Permission checking when sending SMS . . . . . 23

4.1 Using vibrator settings as covert communication channel . . . . . 26

5.1 Example of SCanDroid flow policy violation . . . . . 36
5.2 SCanDroid flow extraction for single application . . . . . 38
5.3 Combining SCanDroid application flows . . . . . 40

6.1 Policy enforcement moments . . . . . 51
6.2 Linux socket state machine . . . . . 57
6.3 Socket classes . . . . . 60
6.4 Application accepting an incoming connection . . . . . 62

7.1 Application connecting to the Google homepage . . . . . 70

# 1

# Introduction

The foundation for digital mobile phone technology as we know it today has started in 1982, when the Confederation of European Posts and Telecommunications (CEPT) formed the Groupe Speciale Mobile (GSM) with the goal to design a pan-European mobile telephony technology [13]. By the year 2000, the first commercial General Packet Radio Service (GPRS) services were launched, allowing mobile devices to interact with the global internet using a packet switched IP interface[1]. Faster packet switched interfaces were introduced in the years that followed.

Alongside the evolution of GSM technology and ubiquitous mobile IP connectivity, device manufacturers succeeded in creating smaller, more powerful and less power-hungry processing and memory components. Also on the software side, advances were made, eventually allowing third-party applications to execute on the device hardware, creating the possibility for end users to extend the functionality of a mobile device after purchase. According to most sources, this post-purchase software installation capability is what differentiates a smartphone from ordinary phones [1, 18, 29].

One of the smartphone platforms that recently gained popularity is Android. Android was initially developed by Android, Inc., a company co-founded by Andy Rubin and acquired by Google, Inc. in July 2005 [7]. The Android distribution and the foundation of the Open Handset Alliance – a consortium of hardware manufacturers, software companies and telecom operators, publicly led by Google, Inc. – were announced in November 2007.

Towards developers, Android is explained as a software stack for mobile devices that includes an operating system, middleware and key applications (see “What is Android?” in [2]).

For security, Android employs a permission model that enables applications to request permission to perform actions that may access private information, use

[1] GPRS actually offers a wireless packet switched interface, capable of carrying any packet switched protocol, like IPX or X.25. Most operators only deploy IP over GPRS, since IP is used for accessing the Web and carrying internet mail.

the mobile network in ways the may incur a monetary cost to the user or may otherwise cause harm. The model is based on a static set of permissions that an application states that it needs to function and the user can either choose to grant those permission or abort the installation of an application.

## 1.1 Research question

*How can the permission model employed by the Android smartphone platform be improved, such that some known weaknesses are fixed and the existing protection and usability are preserved?*

## 1.2 Subquestions

In order to formulate an answer to the research question stated in Section 1.1, several subquestions have been formulated. These subquestions are constructed in such a way that these can be researched in an ordered fashion. The answers to those questions are given throughout this thesis and result in an answer to the research question.

### Q1. How is the current Android permission model designed?

The current permission model used in Android has been designed to prevent applications from performing actions that the user doesn't consent to. Examples of resources that are protected by the permission model are access to stored text messages and contacts, access to the internet and actions that incur a monetary cost to the user, like sending text messages or initiating phone calls. As a starting point, this subquestion aims to understand the currently implemented permission model and is covered by Chapter 2.

### Q2. Which platform component(s) contribute to the implementation of the permission model?

Given the design of the permission model, how is it implemented in the Android platform? The platform consists of a large number of components that fulfill functional and non-functional requirements, but most components don't take an active role in enforcing the permission requirements. For a thorough understanding, this subquestion aims to provide an overview of the implementation components that take part in implementing the permission enforcement requirements and is covered by Chapter 3.

### Q3. Which vulnerabilities are known to exist in the current model?

In order to improve the permission model, it is required to have an overview of the vulnerabilities that must be addresses by the proposed improvements. Researchers found several weaknesses in the current permission model and those have been published in existing literature. This subquestion aims to identify the known vulnerabilities and is covered by Chapter 4.

### Q4. Which improvements have already been developed?

Existing literature already describes several improvements that can be implemented to enhance the current permission model. This subquestion aims to give an overview of the existing improvements previously proposed by researchers and is covered by Chapter 5.

### Q5. How can the permission model be improved to fix the identified vulnerabilities?

By identifying the weaknesses that remain unresolved in existing literature, a list of necessary improvements has been formed. By enhancing the model, addressing the known vulnerabilities, a final improved model can be designed, which will be the main contribution of this thesis. This final model has been used to answer the main research question defined in Section 1.1. This subquestion is covered by Chapter 6.

## 1.3 Thesis overview

This chapter finishes with an overview of related projects that will not be referenced later in this thesis, but that are relevant to this topic. Chapter 2 will present the Android application model and the way applications are secured against each other. The implementation details of the intercomponent communication framework and the way permissions are enforced is presented in Chapter 3. Chapter 4 presents some vulnerabilities that have been identified in existing research and Chapter 5 presents some improvements that have been proposed to overcome some of these vulnerabilities. The contribution of this thesis is presented in Chapter 6, which proposes a solution that allows a more fine-grained policy for Internet access to be enforced by the Android platform. Finally, Chapter 7 relates the proposed work to the research question and subquestions that have been posed in Sections 1.1 and 1.2 and lists some topics for future research.

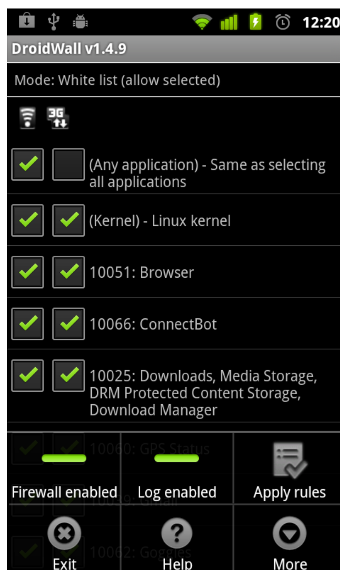



Figure 1.1: DroidWall configuration activity that can be used by the user to select what applications are allowed to communicate [20]

## 1.4 Related work

The security properties of the Android platform – as discussed in Section 2.3 – are only scarcely documented and spread over several sources. Most information can be found in the Android developers guide [2], but in any case, all documentation is limited to textual descriptions. Shin, et al. [22, 23] created a model that captures some important elements of the permission model and have used the Coq[2] proof assistant to prove certain security requirements they extracted from the Android documentation. Although their model isn't complete – e.g. they haven't modeled the fine-grained extension of content provider permissions – they already demonstrated their approach is capable of revealing flaws in the Android permission model.

### 1.4.1 DroidWall for Android

The DroidWall [20] project for Android has managed to implement a user interface to the `netfilter` packet filtering subsystem that is part of the Linux kernel and follows the principle described in Section 6.1.2. As can be seen in Figure 1.1, DroidWall enables the user to enable or disable connectivity for individual appli-

[2]Coq is a formal proof management system that provides a formal language to write mathematical definitions, executable algorithms and theorems.

Figure 1.2: Firewall iP prompting the user for network access [14]

cations and allows to define a different policy when connecting over Wi-Fi or a connection to the mobile operator.

Although technically possible, the DroidWall project does not allow the user to create a more fine-grained policy as will be possible with the implemented protection proposed in Chapter 6.

### 1.4.2 Firewall iP for iPhone

Compared to Android, the iPhone platform introduced by Apple, Inc. has a different security model. On genuine iPhone devices, native applications can only be installed when they are published in the App Store. An extensive screening procedure is implemented to ensure no malicious application should be available in the App Store.

Native applications from other sources can only be installed after removing the technical limitations using a procedure known as "jailbreaking". This procedure involves using vulnerabilities in the operating system in order to be able to replace system components without triggering other security controls.

One of the applications that can be installed on a jailbroken iPhone is Firewall iP [19]. The application allows the user to create a fine-grained policy to restrict the communication capabilities for applications. It can also request the user for permission when an application requests a new connection that is unknown in the security policy, as can be seen in Figure 1.2.

No documentation is available that describes how Firewall iP enforces the policies, but the FAQ page of the Firewall iP website [19] reveals that a library with

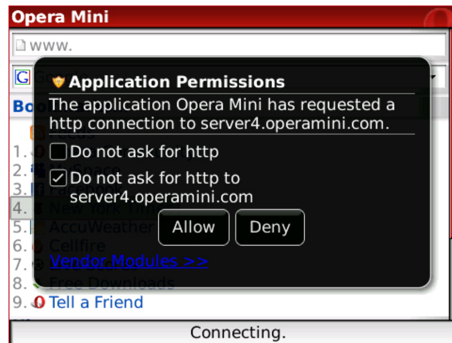



Figure 1.3: BlackBerry prompting the user for network access [10]

the policy enforcing code is injected into the process of applications using MobileSubstrate [11]. MobileSubstrate allows developers to inject code into the address space of other applications to replace the existing implementation of specific classes.

Given the information available on the FAQ page of [19] and the functionality offered by MobileSubstrate, it can be assumed that the functionality of Firewall iP is implemented in a small library that is injected into other applications, such that it enforces a network policy when applications request establishing a new connection.

Under this assumption, it seems possible to bypass the protection mechanism. Because the code that is used to implement the policy enforcement resides in the address space of the application that it protects, the application is probably able to either modify the code (by overwriting the code in memory) or bypass it by again replacing the implementation of these classes with an implementation that directly invokes the original code.

### 1.4.3 BlackBerry device firewall

BlackBerry uses a permission system to allow users or the IT department of a corporate environment to determine what resources applications are allowed to access and what privileges they have to perform operations that may be harmful to the user. This permission system closely resembles the one in Android, with the main difference being that with BlackBerry the user is able to alter the set of permissions that is granted to an application, much the same as Apex makes possible with Android (see Section 5.4).

For several permissions, BlackBerry allows to redirect the communication to a built-in firewall component, which enables the user or IT department to enforce a more fine-grained access policy (see Pages 337–340 of [15]). This built-in firewall

can be used for internet communications, which contains three modes that can be applied to an application: always allow, always deny or prompt, with prompt being the default setting. When prompt mode is enabled, the system will freeze an application that opens a connection to another host and request the user to decide whether or not to allow this connection request, as can be seen in Figure 1.3.

2

# The Android permission model

The first step towards understanding and enhancing the Android permission model is to see how what the general application model looks like and what security constructions have been implemented. This chapter will focus on the first subquestion of this thesis:

*Q1. How is the current Android permission model designed?*

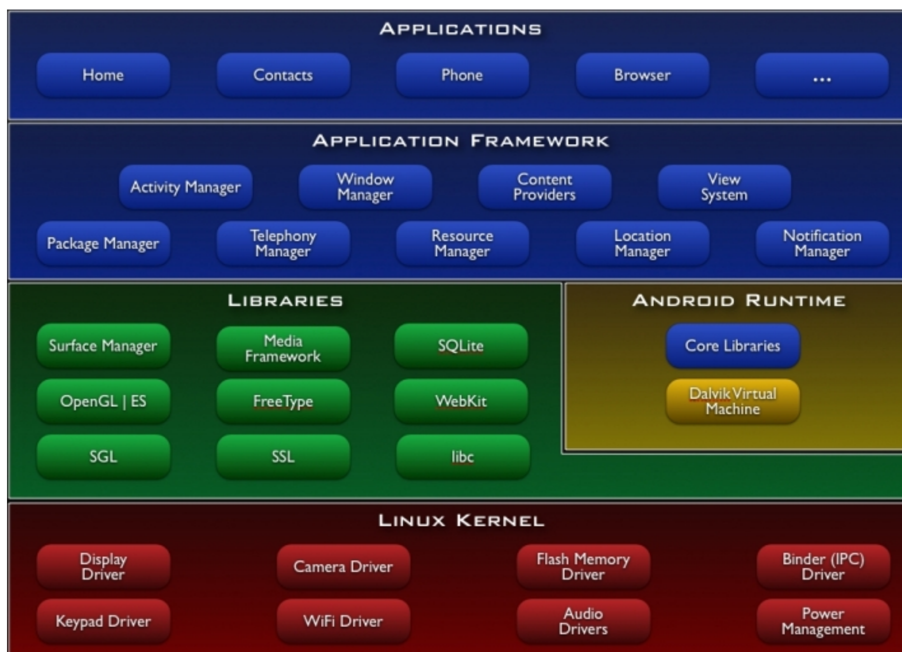



Figure 2.1: System architecture of the Android platform [2]

## 2.1 Android architecture

Figure 2.1 contains an architectural overview of the Android platform as it is published in the Android Developers Guide [2]. The components can be grouped in five classifications that make up the Android platform:

**The kernel** is the lowest level in the architecture and is currently a standard Linux 2.6 kernel. The kernel contains drivers to operate the hardware of the device and is responsible for supplying low-level functionalities like threading and low-level memory management. One of the Android-specific components added to the Linux kernel is the Binder, which is used by the platform to facilitate communication between application components, see Section 3.1.1.

**A set of C/C++ libraries** is running on top of the kernel. Most of these libraries are existing open source libraries, tuned for execution on embedded Linux-based systems.

**The Android Runtime** contains the Dalvik Virtual Machine (see also Section 2.3.2), that implements a dialect of the Java Runtime Environment, allowing Android applications and the application framework to be implemented using Java technology.

**The application framework** provides the API (Application Programming Interface) that allows applications to interact with the device hardware and with each other. Core applications shipped with Android, like SMS and contact applications, use the same framework API that is available to third party developers, allowing all developers to build powerful and integrated applications.

**Applications** run on top of the application framework and are mostly written in Java. Section 2.2 further describes what components make up an Android application.

The remainder of this thesis mainly focuses on the Binder kernel driver, the Dalvik virtual machine, some application framework components and the applications compartment, since these are specific to Android. The other components are rather common in Linux-based systems – including servers and desktop computers – and are therefore out of the scope of this thesis.

## 2.2 Applications

Android applications are mostly written in Java[1] and are built up using components that can be of four types: activities, services, broadcast receivers and content providers [2, 4, 9]. To explain these component types, an example application is used that can track the location of friends and notify other applications about friends that are nearby.

**Activities** are the fundamental concept that allows interaction with the user. At any point in time, a single activity is visible on the screen. A simple application might have only a single activity, while more complex applications consist of several activities that, together, form a user interaction model. Normally, an activity is started at the moment that it needs to interact with the user and it is stopped as soon as it is not visible on the screen anymore.

The example application might contain an activity that displays all friends in a list, maybe including their current location. Another activity in this application can be a settings screen, allowing the user to modify the behavior of the application. Yet another activity can be a form to add a new friend to the application.

**Services** are components that are not visible to the user, i.e. that don't have a user interface. The lifetime of a service is therefore not automatically linked to the visibility of user interface elements, but controlled by the operating system in response to the need of the service and available resources. Applications can bind to a service that is already running in order to communicate with them. When a service isn't already running when an application binds to it, the Android platform will start up the service.

In the example application, the component that receives location information about friends and updates this location in the local storage and/or informs other application about friends getting nearby or farther away is a service. It runs in the background and doesn't interact with the user itself.

**Broadcast receivers** also don't have a user interface, but are designed to react to events, such as a change in timezone or a battery running low. Broadcasts can be sent by both the platform and components or third party. Broadcast receivers may bind to services in order to start an operation after receiving a broadcast or may start an activity to initiate interaction with the user.

Again using the above example, a broadcast receiver can be a component in a third party application that will be informed when the location of friends

[1] The Native Development Kit (NDK) allows Android applications to be written partly or entirely in a lower level language, enabling applications that have higher performance requirements than can be offered when running code in a virtual machine.

changes. When the service notices this event, it sends a broadcast, which will trigger the broadcast receiver.

**Content providers** provide an interface to applications to access data provided by the content provider application. Often, a content provider uses the file system or an SQLite database to store the data, but any method that's appropriate for the type of data is acceptable.

The example application needs to store the tracking details and current location of friends. This information is probably stored in a database. To give other components in the example application – or even components in third party applications – access to this information, a content provider can be used.

When one of the components of an application needs to run and the application is not already started, a Linux process running the Dalvik VM is created and the component will run in this process. By default, all components of a single application run in this process, but any application is free to start new processes and run arbitrary Java and non-Java code.

## 2.3 Android security model

Android incorporates several mechanisms to prevent applications to adversely affect each other's operations [2, 4]. By running applications in a sandbox, the only way for two applications to communicate is through explicitly shared resources. Access to those shared resources is guarded by a permission model. Applications don't have access to any resource by default, all required permissions must be requested at install time.

### 2.3.1 Application sandbox

The application sandbox is implemented in Android by running each application in an individual Linux process and allocating a unique user ID for each application. The Linux kernel manages access to files and other objects in the file system using the standard access control procedures used in UNIX systems[2]. As a result of this, files created by one application are not readable by other applications.

One exception to this general rule exists: two applications that are created by the same developer can request a shared user ID. The requirement that the applications are created by the same developer prevents malicious developers to

[2] In UNIX, for every object in the file system, an owner, a group and a file mode is stored. The file mode contains permissions bits, specifying read, write and execute access permission for the principles "owner", "group" and "other" [27].

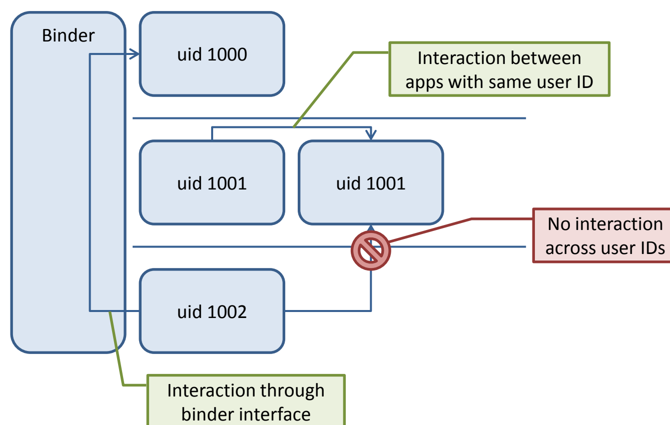



Figure 2.2: An example of a set of four installed applications in a sandboxed environment. Two applications created by a single developer have requested to share a user ID and are therefore able to exchange information using the file system. Arrows indicate interaction paths; the direct interaction between the application with uid 1002 and an application with uid 1001 is not possible. The only way for the other applications to communicate with components outside of the sandbox is by using the binder interface (see Section 3.1).

circumvent the permission system by requesting a shared user ID with a sensitive application. To ensure that the applications are created by the same developer, Android verifies that both applications are signed using the same certificate.

Application components running in different Linux processes are able to communicate by using a standardized inter-component communication framework. The ICC framework employed by Android builds upon the OpenBinder framework [17]. A more technical description of the binder framework can be found in Section 3.1. Figure 2.2 shows how applications are able to interact using the binder framework.

### 2.3.2 The role of the Dalvik VM

One of the main requirements of the Dalvik VM is to execute bytecode under the stringent memory requirements imposed by embedded systems; this has therefore been one of its main design goals [6]. The bytecode that Dalvik operates on is not identical to standard Java bytecode and therefore the Dalvik .dex file format is incompatible with standard Java .class files. The Android SDK includes the *dx* tool [2], that allows developers to convert .class files to .dex files.

While it may be tempting to assume that Dalvik acts as a security sandbox, this isn't actually the case. Quoted from the "Security and Permissions" sections of the Android Developers Guide [2]:

> The kernel is solely responsible for sandboxing applications from each other. In particular the Dalvik VM is not a security boundary, and any app can run native code (see the Android NDK[3]). All types of applications — Java, native, and hybrid — are sandboxed in the same way and have the same degree of security from each other.

Despite Dalvik not being a security mechanism, relying on the Java programming language for application development enforces type-safety and prevents common riskful programming mistakes, like buffer overflows, memory leaks and decreases the possibility of remote code execution in general. Since applications may use non-Java code, it should be noted that these programming errors can still occur in Android applications. Vulnerabilities might also exist in the C/C++ system libraries and those might be exposed through official API's towards Java applications.

### 2.3.3 Declarative permission model

Besides user ID sharing, several communication mechanisms are available for applications that need to interact. These mechanisms are defined by the Android platform and are protected by a permission checking system. The permissions that an application needs are declared in a manifest file, which is part of the application package. Examples of permissions defined by Android are the `INTERNET`[4] and the `SEND_SMS` permissions.

During installation of the application, Android informs the user about the declared permissions, such that the user can decide whether or not to proceed with the installation. In the current model, it is not possible to install an application and grant or deny individual permissions. The only way to deny a permission is by aborting the installation procedure. Consequently, the only way to revoke a permission is by uninstalling the entire application.

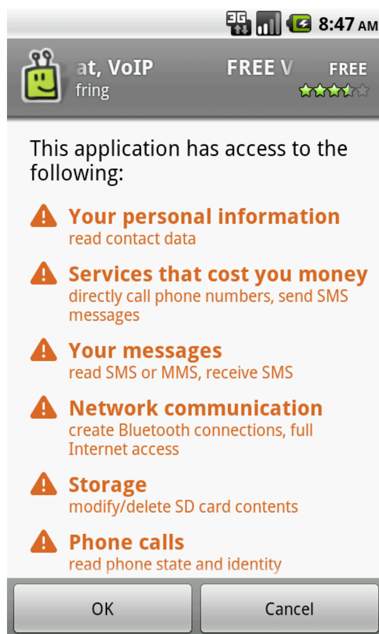
### 2.3.4 Implicit permission checking

Besides explicit checking of granted permissions by application code, the Android platform implicitly verifies whether an application possesses a certain permission on a specific set of inter-component operations. The following overview describes what operations invoke the implicit permission checking [2]:

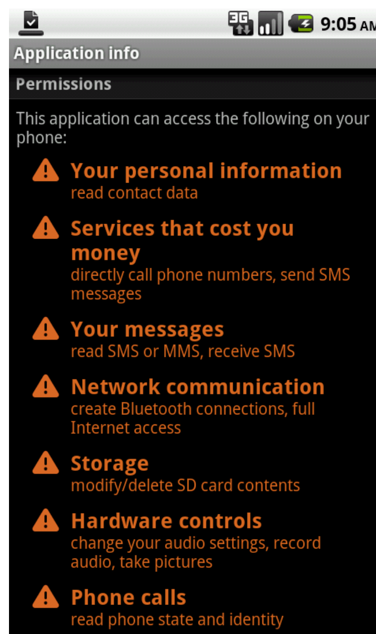
**Activities** can be started by components by invoking the `startActivity()` method of the `Context` class or the `startActivityForResult()`

[3] The Android Native Development Kit

[4] The full name of the permission is `android.permission.INTERNET`, but for the sake of brevity the `android.permission.` prefix is omitted in this thesis.




(a) Application installation screen




(b) Application info screen

Figure 2.3: Permissions screens for the Fring application. Screen (a) is displayed when the user selects the application in the Android Market, such that he can make the decision whether or not to continue the installation. Screen (b) is displayed when the user opens the application information activity in the settings part of the Android system. Notice that the `SEND_SMS` permission is displayed as 'send SMS messages' under the 'Services that cost you money' header.

method of the `Activity` class. These methods check whether the caller has the required permissions and when this is not the case, a `SecurityException` is thrown.

**Services** can be managed by invoking the `startService()`, `stopService()` and `bindService()` methods of the `Context` class, which all check whether the caller possesses the required permissions and throw a `SecurityException` when this is not the case. Note that after the `bindService()` successfully returned, the caller can directly invoke exposed methods on the service, without implicit permission checking by the Android platform. However, services can still use explicit permission checking by invoking permission checking methods on the `PackageManager` or `Context` objects.

**Broadcast receivers** can specify the permission that the sender of the broadcast must have. Broadcasts that are sent by components that don't have the specified permission are silently dropped by the platform. The other way around,

senders can specify the permission that receivers must have. Receivers that don't have the specified permission are not informed about the broadcast. Since broadcast delivery is asynchronous, no exception is thrown in the case of missing permissions, the broadcast will just be ignored silently.

**Content providers** can be used to read from and write to content storages using methods available in the `ContentProvider` class. The available communication methods are protected by two distinct permissions: one permission that is required for reading data using the `query()` method and another permission that is required for writing using the `insert()`, `update()` and `delete()` methods. Android enables components that have access to a content provider to extend those permissions to other components in a fine-grained manner, which is described in more detail in Section 2.3.5.

### 2.3.5 Content permissions

In the packaged manifest file, an application can declare it needs permission to use a content provider. This declaration is for either the read permission or the write permission or for both. Once the user grants these permissions during installation of the application, the permission is valid for all data that is managed by the content provider. This application may now request a component in a different application to perform a task on a specific item managed by this content provider, e.g. an e-mail application may declare it needs read access to the attachment content provider and it requests the image viewer to display an image that was received as an attachment to an e-mail.

When using a purely declarative permission model, a dilemma occurs for the developer of the image viewer. If he chooses not to declare that the component needs the read permission, it won't have access to the attachment content store and therefore the component won't be usable to the e-mail application. However, if the developer chooses to declare the read permission requirement, the component will have access to all attachments stored by the content provider. This last option may allow yet another component to abuse the image viewer component to access the attachment content provider without declaring permissions to it. Even when the developer decides to declare it needs read permissions to content providers, he probably doesn't know which content providers the component might need access to, now or in the future.

For this scenario, the Android platform allows an application that has permission to use a content provider to extend this permission to the component that it requests to act on a content item, such that only access to this single content item is granted. Figure 2.4 shows how this applies to the given example: the e-mail

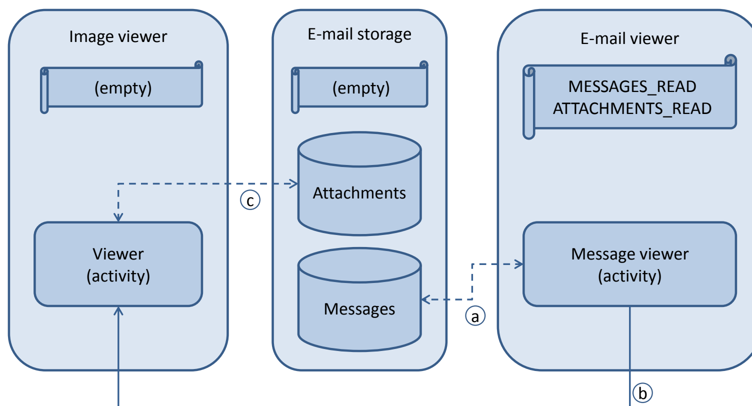



Figure 2.4: Extension of content permissions. The image viewer application doesn't possess any permission to access the attachment content provider. The message viewer activity accesses the message store (a), using the MESSAGES_READ permission granted to the application. When it starts the image viewer to display an image from the attachment content provider, it can pass on the required ATTACHMENTS_READ permission for exactly this single image (b). During the period that the viewer activity is active, the image viewer application can read this single image (c), but doesn't have access to other items managed by the attachment content provider.

application can extend its read permission to the image viewer on the particular attachment that it requests the image viewer to display.

## 2.4 Conclusion

This chapter has introduced the Android platform, which is based on the Linux kernel. Individual applications are mostly written in Java and each run in their Java virtual machine, running in its own process. Since application have a unique user ID, the process boundaries actually act like sandboxes, preventing applications to influence each other directly. Applications can use functionality offered by other applications and by components of the Android platform through a standard interface, which enables the system to enforce a declarative permission model. Together, the sandbox and the permission model ensure security in such a way that applications are unable to perform actions that are not permitted by the user.

# 3

# Implementation details of the permission model

After forming a high-level understanding of the Android application model and security constructs, which are explained in Chapter 2, this chapter focusses on the next subquestion of this thesis:

> *Q2. Which platform component(s) contribute to the implementation of the permission model?*

While the knowledge required for application developers is discussed in the Android Developers Guide [2], the implementation details are not extensively discussed in documentation and other research papers. Therefore, most of the knowledge discussed in this chapter is based upon source code analysis. Instructions to obtain the source code are published by the Android Open Source Project[1].

## 3.1 Inter-component communication

Application components running in different Linux processes are able to communicate by using a standardized inter-component communication framework. The ICC framework employed by Android builds upon the OpenBinder framework [17], which has originally been designed and developed by Be Inc. and more recently by Palm, Inc. On top of this framework, Android has built two layers: the first transforms low-level communication with the kernel driver into a communication paradigm in the Java environment and the second subsequently wraps the developer-unfriendly generic paradigm in a clean object oriented interface. Figure 3.1 shows the Java-to-Java path of an inter-process method invocation.

### 3.1.1 ICC kernel driver

The Linux implementation of the OpenBinder framework that has been incorporated into Android contains several components that together allow Java objects to communicate across virtual machine and process boundaries. At the lowest level,

[1] See `http://source.android.com/`

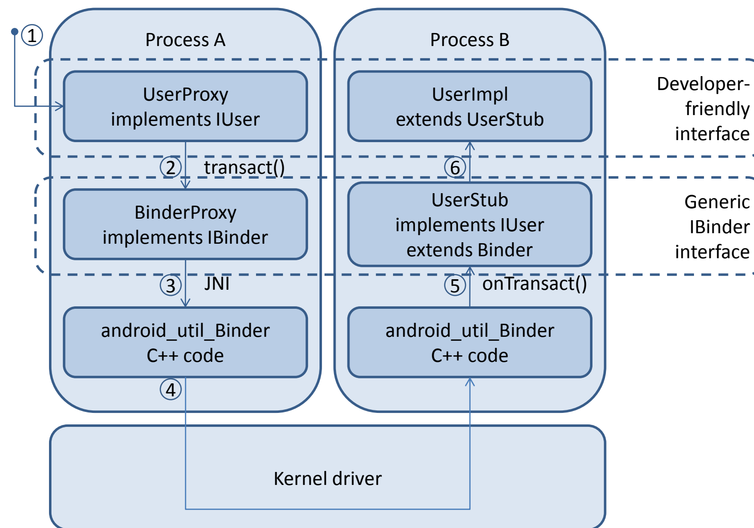



Figure 3.1: Overview of inter-process communication between Java objects in two separate processes. The class names in this figure are the result of an `IUser.aidl` interface definition. When a method is invoked on the `UserProxy` object in process A (1), `transact()` is invoked on the `BinderProxy` (2), forwarding the `Parcel` through the kernel driver towards the other process (3,4). The `onTransact()` method (5) unmarshalls the parameters in the received `Parcel` and invokes the appropriate method in the `UserImpl` class (6), which must be implemented by the developer of the application running in process B.

a binder driver is included in the kernel. This driver is exposed to userspace processes by means of the `/dev/binder` filesystem object, which allows interaction with the kernel driver using `ioctl`[2] system calls.

The binder driver is responsible for passing messages across processes. Since those message may contain references to objects that can only live in a single process, the driver also manages those references and rewrites messages to ensure the receiving process knows how to interact with these objects. When interacting with the driver, objects that live in the address space of the interacting process are referred to by using a pointer, while objects that live in another address space are referred to by using an opaque handle, which is just a unique integer serving as a key into a mapping table.

Before a message is passed from the driver to a process, any object reference is translated into handles that are known inside the receiving process. At this point in time, the driver may allocate a new handle if the given object hasn't been communicated towards the receiving process before. Likewise, any handle is translated into either a pointer to the object if the referred object lives in the address space

[2]In UNIX, `ioctl` operations allow userspace processes to interact with kernel objects beyond the limitations of the standardized file operations like those offered by the `read` and `write` system calls.

of the receiving process, or into a handle that is valid for the receiving process. Consequently, a handle can only be interpreted in the context of a single process and a process is therefore unable to obtain a handle to an object that is not sent to it.

### 3.1.2 Interfacing the kernel driver with Java

Java code running inside the virtual machine cannot directly access non-Java libraries or system calls. To interface the Java environment with functionality in the non-Java environment – i.e. native code – the Java Native Interface (JNI)[3] is available. Using a standardized interface, developers can create native code which interacts with the virtual machine. Using JNI, several communication paths become available:

- Native code can access the values of members fields of Java classes by calling functions like `GetObjectField` or `SetIntField`;
- Native code can invoke Java methods by calling functions like `CallBooleanMethod`, which will run the selected Java method in the VM; and
- Java code can invoke methods that have been marked with the `native` keyword in the class definition, which will instruct the VM to call a defined function in native code.

The core Java interface for inter-process communication is the `IBinder`, which has two major implementations: the `Binder` class for "incoming" communication and the `BinderProxy` class for "outgoing" communication. The core of the `IBinder` interface is the `transact` method, which accepts a numeric argument specifying the operation to execute and two `Parcel` objects to carry the input and output messages.

In the `BinderProxy` implementation, the `transact` method is invoked through JNI and implemented in the C++ file `android_util_Binder.cpp`. Eventually, this causes the contents of the first `Parcel` to be transferred to the kernel driver, which rewrites the object references and passes the message on to the receiving process.

In the receiving process, the message is reconstructed into a `Parcel` and passed to the `onTransact` C++ function, which uses JNI to invoke the `execTransact` method on a `Binder` object. At this point, the `Parcel` object

[3] JNI is developed by Sun as part of the Java standard and is therefore not specific to the Android platform or the Dalvik VM.

is transferred into the Java environment and the `onTransact` method is invoked, which contains the implementation to perform the requested operation.

When the object resides in the invoking process, the `transact` method is directly invoked on the `Binder` implementation, which update the `Parcel` objects, such that they're in the same state as if they were just received through JNI and invokes the `onTransact` method. With this construction, developers can interact with local `IBinder` objects in the same way as with remote `IBinder` objects.

### 3.1.3 Developer-friendly communication layer

While the `IBinder` interface described in Section 3.1.2 is the central communication point with other processes, it requires flattened messages to be exchanged using `Parcels`. This requires communicating objects to marshall and unmarshall information, which is a tedious and potentially error-prone task when done manually by developers.

The Android SDK (Software Development Kit) contains the code generator tool `aidl` (see the AIDL appendix in [2]). The developers defines the interface between the components in an interface definition file, which the `aidl` tool uses to generate two Java classes: a proxy and a stub.

The stub is an abstract[4] class and extends the `Binder` class. Its `onTransact` method unmarshalls the `Parcel` that it receives, invokes the appropriate method on the implementation class and marshalls the results back into a `Parcel` object. The developer now just needs to extend the stub class and implement all abstract methods, which are the ones defined in the aidl interface definition.

The proxy is a class that implements all methods defined in the aidl interface definition. All those implementations marshall the invocation arguments into a `Parcel` object, which is passed to the `transact` method of an `IBinder` object. The `transact` method forwards the `Parcel` through native code and the binder kernel driver towards the remote object, where a stub performs the method invocation on the actual object. The reply `Parcel` object is forwarded in the opposite direction. The proxy unmarshalls the data from this object and returns control to the invoker. A developer can now just invoke a method on the proxy object to cause a method invocation on the remote object.

## 3.2 Enforcing permissions

Section 2.3.3 introduced the declarative permission model implemented in the Android platform. Enforcing those permissions is done by the `PackageManager`

[4]In Java, abstract classes may declare methods without an implementation. Abstract classes can therefore not be instantiated, but must be extended by a subclass which contains at least an implementation for the abstract methods.

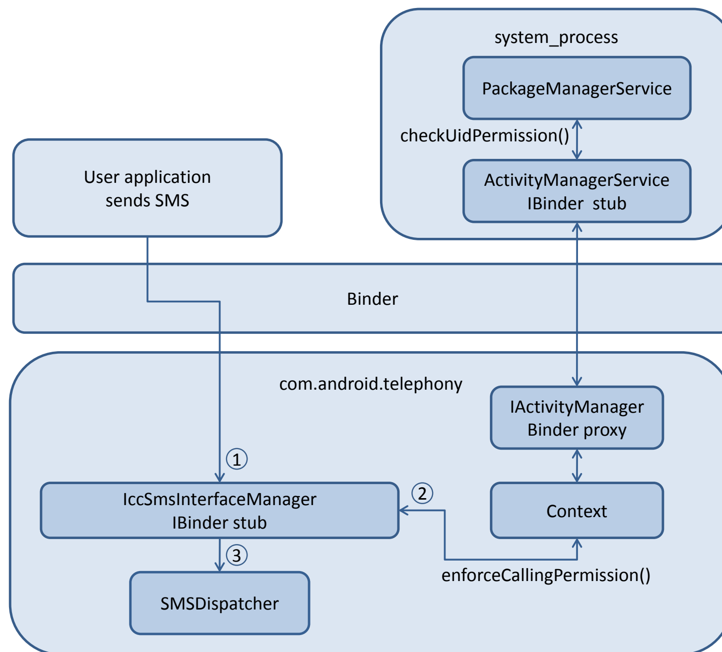



Figure 3.2: Permission checking when sending an SMS message using the `SmsManager` API. The SMS message is submitted to the telephony process using the binder interface (1). Before forwarding the message to the `SMSDispatcher` (3), the SMS service verifies whether the sender possesses the `SEND_SMS` permission by calling the `enforceCallingPermission()` method on the `Context` class (2).

component, which runs in the `system_process` process that has been spawned during the boot procedure of an Android system. The `PackageManager` reads the permissions that are requested by applications from the permanent storage and keeps them in memory. When an application wants to verify whether a certain application has been granted a specific permission, it can invoke the `checkUidPermission()` method through the binder interface. This method returns either `PERMISSION_GRANTED` or `PERMISSION_DENIED` and the developer of the service must ensure that the verdict is correctly used when deciding on how to proceed.

When an application invokes a method on an object in a remote process, the implementation of this method can easily verify whether the calling process has been granted a certain permission by invoking the `checkCallingPermission()` method or the `enforceCallingPermission()` method on the `Context` class, which passes the user ID of the invoking process to the `checkUidPermission()` method. This relieves the developer of an application of the task to determine

which process has invoked the method and what application the given process belongs to. Figure 3.2 shows the permission checking flow when an application invokes the `sendTextMessage()` method on an `SmsManager` object.

## 3.3 Conclusion

To allow applications to communicate, an interprocess commucation framework based on the OpenBinder framework is used in Android. A kernel driver handles communication of basic data types between different processes and is also capable of passing around references to objects residing in the address space of a single process. A Java Native Interface library forms the glue between the interface to the kernel driver and the object oriented Java environment. Since Android system components don't run in the same process as the application, the interprocess communication framework must be used to invoke those components. The individual components can verify whether the process that invoked an operation has a certain permission before actually performing the requested operation, such that application are unable to perform operations that are not permitted by the user.

# 4

# Known vulnerabilities of the permission model

In Chapter 2, the Android permissions model was introduced and Chapter 3 discussed the implementation details of this model. Several vulnerabilities exist in this model, that have been presented in research papers. This chapter focusses on these vulnerabilities and therefore on the third subquestion:

> *Q3. Which vulnerabilities are known to exist in the current model?*

## 4.1 Permission-free operations

Most operations that might result in harm to the user, annoyance, monetary cost or privacy violations require permissions that the user must grant to an application before an application is allowed to initiate those operations. There are however some operations that have been identified by researchers that can incur a cost, evoke privacy violations, or otherwise cause annoyance and which are not protected by any permission. Therefore, these operations can be performed by any installed application.

One of the earlier reported operations that doesn't require a permission was the fact that arbitrary applications were able to initiate a phone call through the Phone application. Using the documented API[1], initiating a phone call requires the `CALL_PHONE`[2] permission. However, it was possible to instruct the standard Phone application to start dialing a number immediately after activating the application. When the telephony framework verifies the permissions of the caller, it would identify the Phone application as the caller and discover that it possesses the `CALL_PHONE` permission and would therefore proceed with connecting the call [8].

[1] See the Android developers guide [2]

[2] Without the **`android.permission.CALL_PHONE`** permission, an application remains able to open the Phone application and prefill a phone number, allowing the user review the number and initiate the actual calling.

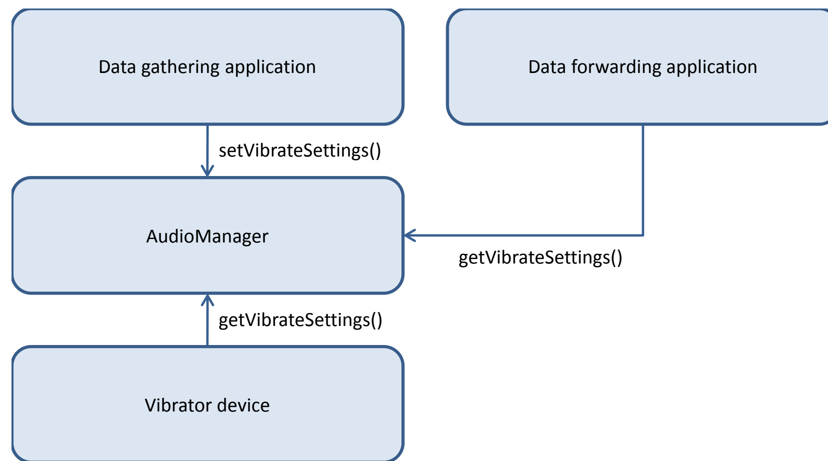



Figure 4.1: Using vibrator settings as covert communication channel. The `AudioManager` maintains the vibrator settings, which the vibrator device uses to decide whether the vibrator hardware is activited. A data gathering application can freely invoke `setVibrateSettings()` to modify the stored information and a data forwarding application – e.g. an application with the `INTERNET` permission – invokes `getVibrateSettings()` to read the information.

Google fixed this issue by adding a permission check in the Phone application, requiring the initial caller to possess the `CALL_PHONE` permission before activating the actual call. Therefore, since Android 1.5 this vulnerability no longer exists.

While the `CALL_PHONE` vulnerability allowed an application to contact premium rate phone numbers, enabling a revenue stream for malicious application developers, some other vulnerabilities – operations that don't easily offer such a revenue stream – are not yet fixed. Two examples are changing the vibrator settings and changing call volume settings. While most settings require a permission to be changed – e.g. `MODIFY_AUDIO_SETTINGS`, `WRITE_APN_SETTINGS` or `WRITE_SETTINGS` – any application can change the vibrator settings and the call volume settings [21].

At first glance, changing these settings would annoy the user at worse, but since the settings changes cannot be traced to an individual application, it will prove difficult for a user to locate the cause of his annoyance. These configuration settings can be read by another application, effectively creating a covert communication channel[3] between applications [21], e.g. by repeatedly turning on and off the vibrator, an application can transmit single bits towards a separate application with the proper privileges to forward the leaked information towards an adversary. Figure 4.1 displays the components that are involved in this sce-

[3] A covert channel is a type of security attack that allows an attacker to transfer information across a boundary in such a way that access control on the channel is not enforced by a security component.

nario; the data gathering application doesn't require any suspicious permissions, while the data forwarding component can be hidden inside any application for which the `INTERNET` or the `BLUETOOTH` permission can reasonably expected to be necessary.

Another method to export data from an Android device without needing a communication-enabling permission is by opening the webbrowser [21]. Although this is not a covert channel, it easily allows a small amount of data to be exporting by encoding it into the URL, which is sent to a webserver to request some content to be displayed. To cover up the data leakage, a malicious webserver could – after storing the received data – redirect the webbrowser to a random webpage, making the user believe the device acted on a accidental click on a banner or link.

## 4.2 Permission-related shortcomings

The vulnerabilities discussed in the previous section are issues that are specific to operations that don't require special permissions; they can be solved by performing a permission check when the discussed APIs are used. However, some more interesting vulnerabilities arise when discussing operations that do require special permission.

### 4.2.1 Permission granularity

Some important decisions must be made when designing a permission-based system, including decisions about the level of granularity to use for permissions in the system. In the design of the permission granularity, some interesting decisions have been made:

**Location information** can be obtained with a high resolution using GPS hardware or with a lower resolution using location information about GSM cells and WiFi access points[4]. To use the former, application need the `ACCESS_FINE_LOCATION` permission, while the latter is protected by the `ACCESS_COARSE_LOCATION` permission. Coarse location information would be sufficient for application like a weather service, while fine location information is necessary for navigational applications. In practice it seems that many application developers have decided to request both permissions [3].

[4]Google has determined the location of a large number of WiFi access points around the world, which can uniquely be identified by the MAC address. Obtaining the location of an Android device using WiFi involves listing the MAC addresses of visible access points and transmitting this list to a Google server, which will return an location approximation based on the stored data.

**Audio recording** can be performed by applications using the MediaRecorder class that have requested the RECORD_AUDIO permission. An application that has been granted this permission can at any time of the day and in any state of the device decide that it start recording audio, without the user getting any notification of this action. In particular, audio recording can commence when a phone call has started, effectively allowing an application to eavesdrop on the communication of a phone call [21].

**Internet access** from an Android device is possible using several communication channels, such as the packet network offered by the telephone operator network, a WiFi wireless network or a bluetooth connection. Different communication channels have different performance characteristics and may incur different monetary costs, e.g. while a WiFi connection may be free to use and offer much bandwidth, the telephone operator network may by much slower and may actually charge the user for data exchange; the monetary burden might even fluctuate depending on the location of the device. Application that have been granted the INTERNET permission are able to communicate with any host on the internet, have no limits on the amount of data they are allowed to exchange and are able to use any communication channel available.

**Content provider** access is controlled by a pair of permissions: one permission for read access and a separate permission for write access. For example, an application requires the READ_SMS permission to be able to read text messages stored on the device, while the WRITE_SMS permission is required to modify the message store [3].

It is interesting to note that location and content permissions are rather fine grained compared to audio recording and internet access. Although the actual permissions are still static – see Section 4.2.2 – the more fine grained permission design for some functionalities gives users more control over what they want to allow an application and what they want to deny.

The INTERNET permission is an interesting example of a coarse grained permission. In fact, over 60% of applications request access to the internet [3]. Some applications even don't need the permission for the core functionality, but use the permission to be able to pull advertisements from the internet in order to generate a revenue stream for the developer of the application. Since the declaration of internet access is this common, users are effectively trained to accept this permission and are blinded to the implications caused by allowing an application access to the network, such as the monetary cost of communication and the possible leaking of information stored on the device.

### 4.2.2 Permission lifecycle

During development of an application, the developer declares the permissions the application needs in a manifest file. Before downloading and installing an application, the user can review the declared permissions and decide whether to proceed with the procedure. In the case the use decides that the permissions the application requests are broader than he is willing to allow, the installation procedure is completely aborted and no application code will be executed.

However, when the user proceeds with the procedure, all requested permissions are granted. As a result of this procedure it is not possible to allow the installation of an application, while only granted a subset of requested permissions [12, 16]. Since most users don't know what operations are connected to each permission, it is difficult to objectively assess the risks involved with granting certain permissions. The choice a user must make is binary: accept a set of unknown risks in order to obtain the needed functionality or abort the installation procedure.

Related to this problem is the fact that permissions that have once been granted to an application cannot be revoked afterwards. When a user determines that an application performs unwanted operations or learns about the implications of granting applications certain permissions, the only available option to limit permissions is to uninstall an entire application [24].

### 4.2.3 Permission mismatching

In [24], Shin, et al. present a flaw in the permission scheme that is partly caused by the fact that permissions cannot be revoked. The flaw they describe enables an adversary to obtain a certain permission $P$ on a third-party application by misleading the user. Three applications are involved in the scenario: a decoy application $A_D$, the exploit application $A_E$ and the protected application $A_P$. The protected application makes an API available to other applications to interact with it, but requires those interacting applications to possess the permission $P$, which is a permission that isn't known to the Android system before installation of the application.

To understand the vulnerability, an example scenario is described. The protected application $A_P$ in this case is an application that stores credit card information in a protected storage and the permission $P$ has identifier `READ_CREDITCARDS`[5]. The exploit application $A_E$ is designed to obtain the credit card data stored by this application, such that it can forward this valuable information to the attacker. Before $A_P$ is installed, the Android system doesn't have

[5]The permission identifier would have a prefix – e.g. `com.visa.android.READ_CREDITCARDS` – but for the sake of brevity it is omitted.

knowledge about the $P$ permission, since this permission will be introduced by the $A_P$ application. Among other information, a textual label is introduced that can be presented to the user, e.g. "Read credit card information".

The decoy application $A_D$ introduces information about a different permission $P'$. The permission is a different permission, but it also has the identifier `READ_CREDITCARDS`. The permission is different, because it introduces the permission with a false label – e.g. "Read system time" – but the system treats it as the same permission, since it has an identical identifier.

Once the $A_D$ application is installed on a device, when installing the exploit application $A_E$, the user will be asked whether or not to grant the `READ_CREDITCARDS` permission. This permission is presented to the user as "Read system time" (note that the internal identifier is not displayed to the user, as can be seen in Figure 2.3 on page 15). Since this textual description sounds harmless, the user probably grants the permissions and finishes installation.

At this point, the exploit application $A_E$ possesses the permission that is identified by `READ_CREDITCARDS`. When at this point the decoy application $A_D$ is uninstalled and the official application $A_P$ is installed, $A_E$ still possesses this permission and is therefore able to read credit card data from the protected application.

Shin, et al. reason that this flaw is possible because of three shortcomings:

1. once a permission has been granted to an application, it cannot be revoked (see Section 4.2.2);
2. as a result, two different permissions with the same internal identifier can exist in a system; and
3. no rule or restriction in naming permissions exists.

### 4.2.4 Using permissions

Applications that possess the correct permissions, can initiate the operations that require those permissions without further approval of the user and often even without the user noticing that such operations take place. For example, recording sound using the microphone of a device can be performed without feedback to the user.

As a demonstration, Schlegel, et al. built an application that listens to `PHONE_STATE` broadcasts to notice when the user initiates an outgoing call and automatically starts recording the audio as soon as it happens [21]. Although such handling of the event of an outgoing call can be desired by the user - e.g. for a call recorder application – the Soundminer application is clearly designed for a malicious purpose.

## 4.3 Executing applications

For applications, there are several ways to become active, either on request of the user or another running application, or in response to certain events. From a security perspective, it is interesting to note that an application can become active without the user explicitly requesting for this to happen or even being able to notice this. For example, applications can declare an interest in BOOT_COMPLETED, which are broadcasted directly after the device finished the system boot sequence [2]. Fortunately, applications need to possess the RECEIVE_BOOT_COMPLETED permission to receive these broadcasts and therefore a user can decide during application installation whether this request is plausible.

However, adversaries can distribute applications that can be started automatically by the Android system when certain events occur. The specific events the applications want to be started can be listed in the manifest file. Only a subset of these events require an application to possess certain permissions; the remaining events can be acted upon without specifying a permission. From the set of events that can be acted upon without having a special purpose, a few can even be specified in the manifest file, which are read by the Android system during installation and will trigger the execution of an application when it isn't already running at the moment the event occurs [21]. The list below presents a non-exhaustive overview of events that may cause the system to execute an application.

**Power-related events** Every application can receive notifications on power changes, e.g. AC power events trigger broadcasts (POWER_CONNECTED and POWER_DISCONNECTED). Battery capacity changes are also broadcast when they pass a certain threshold (BATTERY_LOW, BATTERY_OKAY). No permissions are required to receive those broadcasts [2]. Unless a device is always and only connected to AC power, an application can make use of these events to auto-start itself, without holding any permission and without holding the RECEIVE_BOOT_COMPLETED permission in particular.

**Incoming text messages** When a text message is received by the system, SMS_RECEIVED is broadcast. Applications need to possess the RECEIVE_SMS permission to receive those broadcasts [2].

**Phone state changes** When the state of the phone changes – e.g. on incoming or outgoing calls – PHONE_STATE is broadcast. Applications need to possess the READ_PHONE_STATE permission to receive those broadcasts, but for certain applications the need for this permission might feel plausible to a user [2, 21].

## 4.4 Conclusion

Some vulnerabilities exist in the current permission model, some of which have been presented in this chapter. First, certain device settings can be changed without having any specific permission, which enable some covert channels for malicious applications to communicate. Second, existing research have raised a discussion on the granularity of certain permissions. Third, permissions are granted for the entire lifecycle of an application; it is therefore not possible in the current model to revoke a subset of the requested permissions from an application. Fourth, the way permissions are identified and managed in Android allows a malicious developer to "hijack" a permission identifier. Finally, several permissions are available that allow an application to register itself in such a way that the application is (re-)launched when certain events happen, which may not be obvious to the user.

# 5

# Existing improvements to the permission model

Several vulnerabilities in the Android permission model have been discussed in Chapter 4. Research projects have designed improvements to the existing model. This chapter focusses on these improvements and therefore on the fourth subquestion:

> *Q4. Which improvements have already been developed?*

## 5.1 Changing permission granularity

Section 4.2.1 discusses aspects of the current level of permission granularity. Some of these aspects present permissions that are too coarse-grained to be effective. In particular, the `RECORD_AUDIO` and `INTERNET` permissions are both all-or-nothing permissions. For both of these permissions, several potential solutions have been offered. Also, more general solutions are offered that may be suitable for other permissions.

Audio recording is protected by only a single permission: the `RECORD_AUDIO` permission. When an application possesses this permission, it is allowed to activate audio recording at any time, even during phone calls. Two possible solutions are presented by Schlegel, et al. [21]: phone application isolation and finer-grained sensor access.

Isolating the phone application in such a way that the phone and any other application cannot simultaneously access the same resources. For example, the microphone, speakers and screen cannot be accessed by any application during a phone call, since the phone requires access to those peripherals. The downside of this approach is that this will prevent the functionality of certain applications, like a voice call recorder or an automatic speech translator.

The other proposed solution is defining permissions on a finer-grained level. For example, the `RECORD_AUDIO` permission could be restricted such that it only applies to recording audio in a state where no phone call is active, while at the

same time a new permission can be introduced that allows audio recording during an active phone call.

Barrera, et al. reported that over 60% of applications request the `INTERNET` permission [3]. Probably most of them don't even need access to the Internet for the core functionality, but rely on this permission to retrieve advertisements from the Internet, which are displayed when the application is running to generate revenue for the application developer.

By using finer grained permissions for internet access, users could allow applications to access a specific subset of the internet, instead of allowing an all-or-nothing approach. In [3], Barrera, et al. propose a hierarchical permission scheme that could enable the developer to request access to a specific hostname or set of subdomains, e.g. request access to `*.admob.com` in order to retrieve advertising data.

## 5.2 Kirin security policy invariants

Enck, et al. [8] have designed a framework that allows a device to enforce predefined security requirements before allowing an application to be installed. The framework – named Kirin – is invoked when the system commences the installation of an application package.

Kirin reads the application manifest file and transforms the requested permissions into a set of Prolog[1] facts. Preprocessing steps add additional facts to this set, to make sure all declarative aspects of the Android security model are covered; a complete overview of these facts are listed in Section 4.2 of [8].

Security policies – which are formally a set of invariants that must hold both before and after installation of an application – are encoded as predicates in the Prolog program. The final Prolog program is formed by combining the facts that are derived from the package manifest, the previously derived facts of the Android platform and already installed applications and the predicates that make up the security policies. By feeding this program into a Prolog engine, it is possible to run queries to evaluate the security policies against the information extracted from a to-be-installed application package. If one or more mandatory queries return failure, the system can deny installation of the package.

### 5.2.1 Modeling security policies in Kirin

In the descriptions below, $P$ denotes a security policy, $S$ denotes the set of all subjects, $O$ denotes the set of all object and $R$ denotes the set of all permission

[1] Prolog is a general purpose logic programming language. Prolog programs are expressed in terms of facts and predicates, which are used by a Prolog engine when a query is issued over a given program.

identifiers. In the Android model, the subjects are the applications that can be granted a set of permissions and the objects are components for which permissions requirements can be declared. Note also that two objects are defined for each storage provider: one object that identifies reading from the storage provider and one that identifies writing to the storage provider.

Declarations from the package manifest file are transformed into Prolog facts of the following forms:

$\texttt{requires}(o, r)$ when the manifest file contains a declaration for the requirement of permission $r \in$R to interact with application component $o \in O$.

$\texttt{has_perm}(s, r)$ when the application $s \in S$ possesses the $r \in R$ permission.

$\texttt{contains}(s, o)$ when the application $s$ contains component $o$.

Subsequently, security policies are typed as $P : S \times O \times R \rightarrow \{\texttt{true}, \texttt{false}\}$. The policy operator $P$ is defined as $P(s, o, r) = \texttt{requires}(o, r) \wedge \texttt{has_perm}(s, r)$. Enck, et al. have created three invariant building blocks – called patterns – that can be combined to design more complex security policies. The three patterns are reproduced in the following overview, each with an example that has been taken from [8]. Appendix A of [8] contains a listing of the Prolog program that encodes these three example patterns.

**Simple access control checks** to determine whether an application has the permission required to access a specific object. For example, the following predicate encodes the question "Can application $s$ acquire write access to ContactsProvider?":

$$\texttt{pat}_1(s) = \exists r \in R : P(s, \texttt{ContactsProvider_w}, r)$$

**Mutual exclusion** allows a security policy to disallow an application to obtain two permissions that are regarded undesired to be granted to a single application. For example, the following predicate encodes the question "Can application $s$ be installed such that if it can read from ContactsProvider, then it cannot connect to the network?":

$$\begin{aligned}\texttt{pat}_2(s) = \forall r_1 \in R, \exists r_2 \in R, \forall r_3 \in R :& \\ &\neg P(s, \texttt{ContactsProvider_r}, r_1) \\ &\vee (P(s, \texttt{ContactsProvider_r}, r_2) \\ &\qquad \wedge \neg P(s, \texttt{network}, r_3))\end{aligned}$$

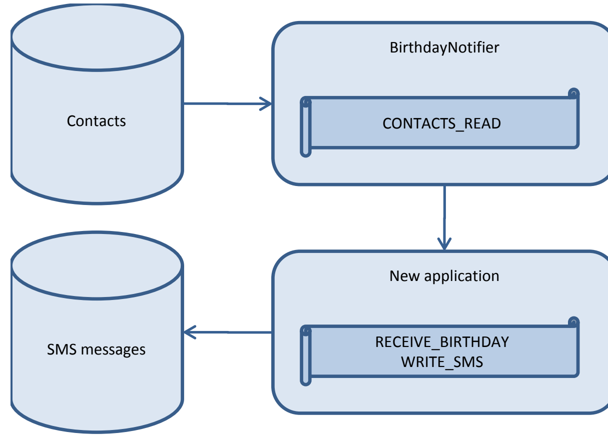



Figure 5.1: Example of flow policy violation. The `BirthdayNotifier` component reads contact information from the contact store. Information about the contacts whose birthday is today is broadcasted to components that possess the `RECEIVE_BIRTHDAY` permission. The new application that is to be installed would be able to obtain contact information from the content provider, through the `BirthdayNotifier` component, without requesting the `CONTACTS_READ` permission. Since this component also requests the `WRITE_SMS` permission, information may flow from the contact store to the SMS store, which might be prohibited by security policy. SCanDroid will analyse the code of the new application to determine whether or not the contact information might flow into the SMS store.

**Rights dependence** can be used to require an application that requests a certain permission, that it also requests another specific permission. For example, the following predicate encodes the question "Can application $s$ be installed such that if it can write to ContactsProvider, then it can also read from it?":

$$\begin{aligned}\mathtt{pat}_3(s) = \forall r_1 \in R, \exists\{r_2, r_3\} \in R :&\\ &\neg P(s, \mathtt{ContactsProvider_w}, r_1)\\ &\vee (P(s, \mathtt{ContactsProvider_w}, r_2)\\ &\quad \wedge P(s, \mathtt{ContactsProvider_r}, r_3))\end{aligned}$$

## 5.3 Information flow analysis with SCanDroid

A flow of information may have unintended security consequences with respect to secrecy and integrity. For example, a flow from store A to store B is unwanted if the data in store A is intended to be secret – i.e. requires a read permission – while store B is readable by any application. This same flow is also unwanted if the data in store B is intended to be trusted – i.e. requires a write permission –

while the data in store A may be tainted. An information flow that might violate secrecy is illustrated in Figure 5.1, which SCanDroid would flag as dangerous.

Fuchs, et al. have developed the SCanDroid framework that performs information flow analysis on an Android application to track how information from one store moves towards another store [12]. This approach allows Android users to determine whether applications that request permission to multiple information stores – e.g. read access to a private store and write access to a public store – will not transfer private information towards the public store.

SCanDroid consist of two logical parts. The first part will be discussed in Section 5.3.1 and processes individual applications, which result in a set of application flows, which are encoded in the form of a mapping from inflow tags to a set of outflow tags. Inflow tags are attached to points where data can flow into an application, such as the result of a content provider query, outflow tags are attached to points where data can flow out of the application, such as content provider update queries. One information flow can thus be described by the interface (inflow tag) where it enters the application and the interface (outflow tag) where it leaves the application.

The second part of SCanDroid will be discussed in Section 5.3.2 and combines the information flows of a set of applications with permission information from the respective manifest files and transforms this information into a set of permission constraints. The set of permission constraints is then combined with the policy constraints and checked for consistency; if no contradictions exist, it is known that – given the policy – no dangerous information flows exist in the verified set of applications.

### 5.3.1 Analysis of a single application

The following is an overview of the architectural components for the flow analysis of a single application, which results in a mapping from inflow tags to a set of outflow tags. This part of the analysis is illustrated in Figure 5.2.

**Bytecode loader** The instructions of an application are encoded in a bytecode format; an instruction set which is not native to the CPU of a device, but is interpreted by a virtual machine to allow an application to run on multiple platforms. The bytecode loader reads the instructions for the class files that make up an application and uses this information to construct a call graph. To facilitate further processing of the call graph, several processing steps are performed such as the replacement of Android API implementations by stubs[2] to decrease processing time for known and trusted components.

[2] In this case, a stub is an implementation of a class or an interface that doens't contain an actual implementation.

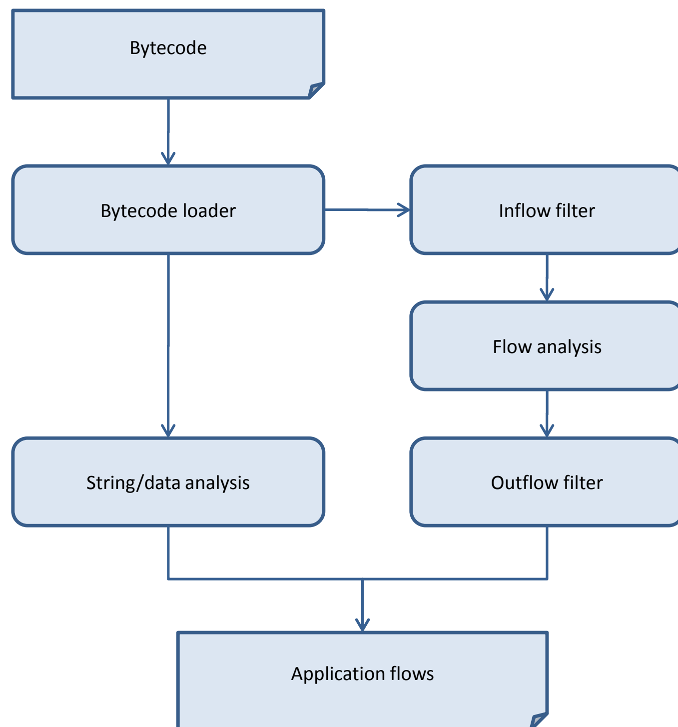



Figure 5.2: Flow extraction for single application. An extensive description of the individual steps is presented in Section 5.3.1.

The bytecode loader also performs pointer analysis on the bytecode instructions and adds context-sensitive information to specific method invocations. This analysis is mostly used to disambiguate data structures returned by methods. In particular, precise information about strings is retrieved, which is then used by the string analysis component.

**String/data analysis** Textual information is processed in several ways throughout the bytecode. For example, Java code that manipulates `String` objects is transformed by the compiler into instructions that manipulate `StringBuilder` objects, as this offers much higher performance and often has a lower memory footprint, especially for complex or extensive manipulations. As a result, bytecode often contains instructions to coerce texts among several different object classes.

The output of the string/data analysis phase is a map from instance keys (identifiers used in subsequent steps) for `String` and `Uri` objects to actual strings.

**Inflow filter** Data flows into an application at certain interfaces, or data sources,

which can be identified precisely. The inflow filter determines all the interfaces where information flows into an application and introduces an instance key for each source, which is then mapped onto a unique tag. The instance keys are used by the flow analysis framework to track how information is transferred between variables, while the tags are used in the output of the application analysis and correspond to the inflow tags discussed in the introduction of Section 5.3.1.

**Flow analysis** The flow analysis phase tracks how information that has entered an application is transferred between variables and methods. The framework makes use of a type system that has been published earlier by one of the authors of the framework [5] and tracks how data flows from instance keys – as assigned by the inflow filter – and local variables to other instance keys and local variables.

The analysis maintains a set of flow identifiers that represent domain elements – e.g. variables, method arguments – that can hold data. The output of the flow analysis phase is a mapping from flow identifiers to a set of (inflow) tags. In other words: the output of this phase records for each domain element which input interfaces can contribute to the information that is stored in the domain element.

**Outflow filter** The result of the flow analysis records information about all domain elements, while only a few of these will flow information out of the components. Therefore, the outflow filter identifies all sinks where information may leave the application an will connect these with outflow tags. By transforming the information available from the flow analysis, the output of the outflow filter is a mapping from inflow tag to a set of outflow tags, which subsequently represents the set of application flows that form the output of the single application flow analysis part.

### 5.3.2 Combining application flow analyses

The analysis of information flows inside a single application, as described in Section 5.3.1, can be performed for an application, without taking other applications into account. The result of this analysis is a set of application flows, which describe how information that flows into an application can flow out of the application. This information is used by the second part of SCanDroid to determine whether dangerous information flows exist when a certain set of applications are installed on a single device.

Following is an overview of the components involved in the second part of a complete SCanDroid analysis, which is also illustrated in Figure 5.3.

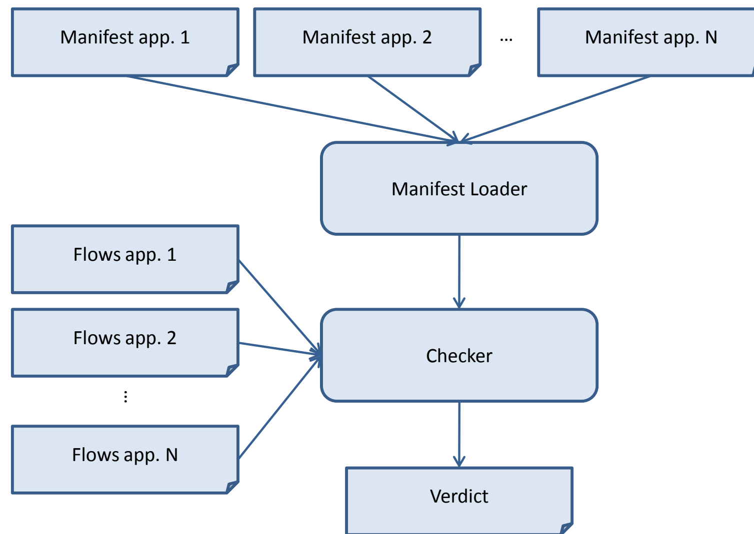



Figure 5.3: Combining application flows

**Manifest loader** Since the manifest file contains information about the components that are part of the application and how these individual components interact with the Android system, SCanDroid needs to obtain this information in order to create a complete analysis. The manifest loader therefore parses the manifest files of individual applications and determines how application components will interact after installation. The manifest loader also attaches the permissions that are enforced and requested by manifest declarations to the corresponding interactions.

**Checker** The checker combines the information from the application flows and the manifest files into a verdict whether or not dangerous information flows can occur, given a policy that dictates what flows are considered dangerous. The checker works by describing a partially ordered set $\geq$, that maintains the reflexivity, transitivity and antisymmetry properties. Elements in $\geq$ are sets of permissions, thus an example of a natural constraint is $\{\mathrm{INTERNET}, \mathrm{CONTACTS_READ}\} \geq \{\mathrm{CONTACTS_READ}\}$.

However, singletons are not necessarily unrelated. The example used in Figure 5.1 demonstrates a situation where installation of the new application introduces the constraint $\{\mathrm{CONTACTS_READ}\} \geq \{\mathrm{RECEIVE_BIRTHDAY}\}$.

A security policy can be expressed as a set of constraints on $\geq$, e.g. the constraint $P_x \not\geq P_z$ can be used to express the fact that the set of permissions $P_z$ is unrelated or strictly higher than $P_x$. The secrecy statement "reading contact information is an unrelated or strictly higher privilege than reading SMS

messages" – which is used Figure 5.1 – can be encoded using the following constraint: {READ_SMS} $\not\succeq$ {CONTACTS_READ}. Using such constraints, also integrity statements can be encoded, e.g. "writing contact information is an unrelated or strictly higher privilege than writing mail attachments" can be encoded as {CONTACTS_WRITE} $\not\succeq$ {ATTACHMENTS_WRITE}.

The checker determines whether a contradiction arises when all policy constraints and all constraints that follow from information flows are combined. When such a contradiction arises, it is known that a flow that violates the security policy might exist. Since SCanDroid analyses information flow of applications individually, this process can be done in an incremental way. Every time a new application is installed, the system gets monotonically more constrained. As soon as a contradiction arises, installation of the new application should be aborted or an existing application should be uninstalled.

## 5.4 Apex permission model extensions

Nauman, et al. [16] have designed the Apex framework that extends the Android permission model to allow more flexibility towards the user. The limitation that Apex is designed to overcome is the fact that the Android design automatically grants all permissions that an application requests. As discussed in Section 4.2.2, the only way to deny certain permissions in Android is to abort the installation of an application altogether. Consequently, the only way to revoke permissions that an installed application already possesses is to remove the entire application from the device.

In Apex, permissions that an application requests are not automatically granted. The framework allows the user to specify conditions that must be met in order for an application to be granted the requested permission. Besides the capability for the user to unconditionally grant or deny a permission, Apex is capable of evaluating condition expressions, such that a user can conditionally grant a permission. Examples of conditions that a user can apply to permissions are:

- Application $A$ is allowed to apply the `SEND_SMS` permission at most five times a day;
- Application $A$ is allowed to use the `ACCESS_FINE_LOCATION` permission only between 9:00 and 17:00;
- Application $A$ is unconditionally denied access to use the `INTERNET` permission; and

- Application $A$ is allowed to use either the `CONTACTS_READ` permission or the `SEND_SMS` permission, but not both.

As described in Section 3.2, the `PackageManager` is the implementation of all permission checking operations. Therefore, the authors of Apex have modified the `PackageManager`, such that after the existing checks have been performed, the `AccessManager` component is invoked. The `AccessManager` is not part of the standard Android framework, but has been added by the Apex authors.

The `AccessManager` invokes the `PolicyResolver`, which in turn uses an XML file to evaluate whether or not the requested permission is currently granted. The XML file contains the permission policy as it has been defined by the user.

## 5.5 Discussion

The previous sections described improvements that have been proposed in various papers. Section 5.2 described Kirin, a framework that allows devices to enforce security requirements based on permission information from manifest files. Section 5.3 described SCanDroid, an information flow analyser that can verify whether the installation of a new application may introduce a dangerous flow. Section 5.4 described Apex, a framework that allows users to conditionally grant permissions that have been requested by applications. This section discusses several aspects of these three improvements.

### 5.5.1 Implementations

The authors of all three frameworks have written a proof of concept to demonstrate the feasibility of their ideas.

Kirin has been implemented as an Android application that runs on a phone and evaluates invariants using only application packages as input, but the authors are working on a tighter integration with the standard package installation system [8].

SCanDroid has been implemented as an application that is not running on a phone, but rather on an external system. The application implements the components that have been described in Section 5.3 and analyses a set of Android applications to generate a set of constraints that result in a verdict [12]. The authors of SCanDroid indicate that the per-application information flow analysis can readily be performed by a third party, certifying the set of application flows that are the result of the first phase. The results of this first phase are unrelated to the actual security policy, so the second part is the only part that needs to be included into the Android system.

The authors of Apex have implemented the framework by adapting the source code of the Android system, such that it is tightly integrated. This was necessary, since it modifies the `PackageManager` to alter the permission checking logic, which is impossible for applications running on top of the framework [16].

### 5.5.2 Effectiveness of Kirin and SCanDroid

The Kirin and SCanDroid frameworks both encode security policies and make sure those are enforced when the user requests the installation of a new application. Kirin casts a verdict based solely on the manifest file of the application, while SCanDroid uses information flow of the new application together with the information flow of all existing applications to cast a verdict. This difference makes that Kirin is by definition much less precise and may report more false positives. The situation may even be worse when the developer of a set of application requests a shared user ID, where Kirin must cast a verdict using the union of the permission sets.

For example, an application that requests a combination of permissions that may be harmful would be flagged as dangerous – given that this combination of permissions has been encoded as dangerous in the manifest file – by Kirin, while SCanDroid would be able to allow this application, given that no actual information flow exists that may violate an encoded secrecy or integrity policy.

A disadvantage of both frameworks is that the policies must be designed upfront and are based on blacklisting. In the case of Kirin this means that a policy writer can create sets of permissions that are considered dangerous, but any combination that haven't been explicitly encoded as dangerous is considered to be safe by default. In the case of SCanDroid, the policy writer must create a set of constraints by deciding what information flows are dangerous. In the same way, any information flow that is not explicitly encoded into the constraint set is therefore considered safe by the framework.

One disadvantage of SCanDroid is that it needs significantly more processing power to reach a verdict, compared to Kirin. However, since the information flow analysis part can be outsourced to a trusted party that can subsequently issue a certification over the resulting information flow data, this disadvantage can largely be addressed.

One problem of SCanDroid – which the authors of the framework did not seem to notice – is the fact that uninstallation of an application does only prevent unwanted information flows in the future. However, before the application had been uninstalled, secret information may already have flown into a store that has not explicitly been marked as secret. For example, consider a situation with three stores $S_A$, $S_B$ and $S_C$ and a policy that disallows any flow of information from $S_A$

to $S_C$. Now, let's say an existing application has moved a datum $x$ from $S_A$ to $S_B$. When the user wants to install a new application that contains an information flow from $S_B$ to $S_C$, SCanDroid would flag the situation and prevent installation. This quote from Fuchs, et al. [12] presents two options to the user:

> ... which means that it is dangerous to let these applications coexist on the same Android device: either the former application should be uninstalled, or the latter application should not be installed.

Although uninstalling the former application will prevent an active information flow from store $S_A$ to store $S_C$ to exist, the datum $x$ might still flow into $S_C$ and therefore violate the policy. Therefore, the only way to definitely prevent policy violations is to ensure that SCanDroid keeps taking the information flows of all previously installed applications into account. In the discussed situation, uninstalling the existing application should not be sufficient to allow installation of the new application.

### 5.5.3 Apex effectiveness

Apex doesn't have the disadvantage that policies must be written upfront, as discussed in Section 5.5.2. Apex allows the user to revoke permissions when an application is installed, so he can review the actual set of permission and can reason about the result that it may have. A disadvantage is that he may forget about certain known dangerous combinations at the time he is installing an application he wants to use. Additionally, the user won't be able to know what information flows exist in the application and could therefore make an uninformed decision.

An advantage that Apex introduces is the fact that users can start using applications with a limited set of permissions and extend the set of granted permissions later. This allows a user to gradually gain trust in the legitimacy of an application, compared to the immediate trust that must be given in the current situation. The other way around is also possible: when a user learns that an application he uses is malfunctioning he can decide to remove unwanted permissions and keep using the application in a restricted environment.

However, Apex has some disadvantages. Most importantly, with the current Android model, application developers expect their application to possess all permissions that have been requested in the manifest file. Therefore, they often lack proper handling of unexpected `SecurityException` exceptions, which is thrown when an operation is requested that the application doesn't have permission to. Since the `SecurityException` is a runtime exception, Java methods are not required to declare this exception and are always allowed to throw it. As a result, it may be assumed that most applications will not catch the exception at

all, allowing the exception to travel down the call stack until it reaches the end of the stack, causing the thread to finish. When this happens in the main thread, Android will display a message to the user that the application has crashed and will stop the application process.

### 5.5.4 Native code assessment

As described in Chapter 2, Android applications are mostly written in Java, but may contain native code written in C or C++. The information flow analysis of SCanDroid is focussed on analysing the Java code and can therefore not be used to analyse applications that contain native code. However, in theory the framework could be extended to take these flows into account. The authors haven't discussed this issue in [12].

Because both the Kirin framework and the Apex framework are not dependent on any code or bytecode, they can be used for native code in exactly the way they have been designed. The only relevant part of an application package that is used by these frameworks is the set of permission that have been requested in the manifest file. Kirin only uses this information to determine whether the installation of an application may proceed. The permission checking modifications that are used by Apex are written in system components that are invoked when privileges operations are requested, regardless of whether such operations are requested from Java or native code.

### 5.5.5 Combining frameworks

Very powerful options become available when two frameworks are combined. Most notably, either Kirin and Apex can be combined or SCanDroid and Apex. When installing an application on a device that has been equipped with the Kirin framework, the installation procedure may be aborted by Kirin when the set of requested permissions violate the security policy. By combining Kirin with Apex, the user would be allowed to deny specific permissions, such that the set of granted permissions does not violate the security policy and Kirin would allow installation.

SCanDroid and Apex can be combined into a powerful partnership in the same way as possible with Kirin. When the installation of a new application would introduce dangerous informations flows, a framework that combines the concepts of SCanDroid and Apex would be able to determine what set of permissions should be denied or revoked to prevent the creation of such flows.

## 5.6 Conclusion

One general improvement has been discussed and three frameworks that are designed to overcome certain vulnerabilities. The general improvement is to introduce a finer grained permission system for some elements, in particular for recording audio and contacting the Internet. The Kirin framework has been presented that allows a security policy to be created, such that a Prolog engine can determine whether installing a new application will violate the security policy. The SCanDroid framework is an off-device framework that is capable of analising the flow of information through an application, such that an on-device component can combine the information flow statistics of the set of installed applications to determine whether a dangerous flow – i.e. one that violates secrecy or integrity of important information – exists. The Apex framework allows the user to define a flexible policy for granting permissions to an application, for example based on time of day or the number of times the application invoked a certain operation. Each of these frameworks introduces a specific enhancement to the Android security model, eliminating some vulnerabilities that exist in the current model.

# 6

# Improving the permission model: the introduction of fine-grained Internet permissions

Chapter 4 presented several vulnerabilities that are present in the current model and Chapter 5 discussed several improvements proposed in existing research. Section 4.2.1 discusses the problem of permission granularity and Section 5.1 proposes a change. This chapter presents a proposal for a more fine-grained Internet permission model as the answer to the fifth subquestion:

> *Q5. How can the permission model be improved to fix the identified vulnerabilities?*

## 6.1 Constraining sockets

The improvement to the Android permission model this thesis presents is to allow more fine-grained control over the resources an application can access via a network connection. As described in Section 4.2.1, applications with the `INTERNET` permission have access to the entire Internet. However, most application only need access to a small set of resources and the principle of least privilege dictates that such application should only have access to those resources it needs to function correctly.

In Linux – and therefore in Android – communication using the Internet Protocol is possible by using sockets. A socket is an object in the kernel and represents a single communication channel. The operations that are available on a socket depend on the type of the socket, the address family in which the socket operates – e.g. IPv4, UNIX domain or bluetooth address family – and the state the socket is in. This chapter focuses on constraining communication with the Internet, so therefore this description will be limited to sockets in the `AF_INET` and `AF_INET6` address families, which are used when communication using IP version 4, respectively IP version 6.

This chapter introduces a method to constrain the use of sockets. To limit the scope of the project, the following requirements are set:

- The behavior of the system must not change for application that possess the INTERNET permission;
- Applications that don't possess the INTERNET permission must only be able to connect to a constrained set of resources specified by a white-listing policy; any resource not explicitly covered by the policy must not be reachable;
- The modifications must only change behavior for TCP sockets; applications without the INTERNET permission must therefore still be unable to use UDP or lower layer protocols, like raw IP;
- Applications that don't possess the INTERNET permission must only be able to connect to ports that are specified by a white-listing policy; an application cannot listen on any port number that is not explicitly covered by the policy;
- Applications that don't possess the INTERNET permission must only be able to receive incoming connections from hosts that are specified by a white-listing policy; any host that is not explicitly covered by the policy must therefore be unable to initiate a connection to the application;
- Application that don't possess the INTERNET permission must not be able to control the local address and port number allocation for outgoing connection, i.e. they cannot use the `bind` system call to bind a socket to a specific address and port.

The goal of these modifications is to allow applications to specify the smallest set of resources they need access to in order to be able to perform their job. For example, a Twitter application may request access to connect to the HTTP port on `api.twitter.com`.

Three alternative solutions have been investigated to meet the set requirements. The following subsections describe these solutions, including a brief discussion about the advantages and disadvantages.

### 6.1.1 Modifying `socket` syscall

To allow applications to connect to a specific set of hosts, the application must be able to create a new socket. Therefore, the permission check on the `socket` system call must be removed and replaced by one or more permission checks in other system calls.

The `socket` system call returns a file descriptor, that the application can use to initiate a new connection using the `connect` system call. Since the application provides the IP address of the remote host to the `connect` system call, the kernel can enforce a policy for outgoing connections at the entry point of `connect`.

For incoming connections, multiple system calls are used to set up a connection. First, an application calls `bind` to inform the kernel of the local address and port number that will be used for communication. Second, the application calls `listen` to inform the kernel to start accepting incoming connection requests – i.e. TCP frames with the SYN flag set – instead of automatically denying all connection requests.

Finally, the `accept` system call is used by the application to obtain a file descriptor for a new incoming connection. This system call may block for an indeterminate time when no incoming connection request is received by the operating system. Enforcing the source host name for incoming connections can be performed either by the network stack as soon as the connection request is received or by the `accept` system call just before the connection is about to be handed to the application.

Restricting the port numbers the application is able to use is a little more difficult. The information is passed to the kernel using the `bind` system call. At this point, disallowed port numbers could be blocked by returning an error message from the system call. However, the `bind` system call can also be used by an application when it wants to enforce a certain port number of IP address is used in outgoing connection requests. As a result, the policy should be enforced at the entry point of the `listen` system call.

Two disadvantages of this method exist that make it hard to deploy this method without breaking compatibility with existing applications:

- Since the `connect` system call will only obtain the remote IP address from the application, the policy can only be based on IP addresses or must be able to link hostnames and IP addresses when evaluating the policy.

- To resolve a hostname, the application needs to contact a DNS server. The policy of every application must therefore include access to the IP addresses of the DNS server. Since the configured address of the DNS server might change – e.g. when switching from WiFi to GPRS – the policy must be updated more often. Additionally, a malicious application can use DNS tunneling [28] to transfer information from and to the device, even without possessing any permission to contact the Internet.

### 6.1.2 Using the netfilter subsystem

The netfilter subsystem is part of the Linux kernel and allows system administrators to implement a network firewall by creating a chain of rules that are evaluated by the network stack for incoming and outgoing IP packets. Every rule in a chain contains a condition – which is evaluated against the packet that is being tested –

and a target. When the packet matches the condition, the target will be executed, which in the most basic cases can be ACCEPT or DROP.

One of the netfilter modules allows a user ID of the process that has created the socket to be tested as part of the condition of a rule. Since Android allocates a unique user ID for each application, this module effectively allows filtering rules to be targeted to specific applications. This method would also allow separate rule chains to be constructed for different network connections, e.g. a WiFi rule chain, a GPRS chain for a mobile connection with the provider and a separate roaming GPRS chain.

An important advantage of this method is that it reuses existing code in the Linux kernel that has been tested in production environments for years. The netfilter subsystem is capable of filtering both outgoing and incoming connections.

However, the disadvantages that have been listed in Section 6.1.1 are also applicable to this method. Another disadvantage of this method is that it will not prevent an application from listening on a port that is not white-listed in the policy. Although netfilter is capable of implementing a list of white-listed port numbers, it is not capable to enforce a per application white-list for port numbers. This means it is unable to white-list a port for a single application, while keeping it unavailable to other applications.

### 6.1.3 Using a trusted socket creator

Another method to allow application to connect to a specific set of hosts is to offload the calls to the `socket` and `connect` system calls to a separate process and to do the same with the combination of `bind`, `listen` and `accept` system calls. With this method, applications that possess the INTERNET permission have unrestricted access to the Internet, while applications that do not have this permission have to rely on the offloading procedure to obtain a connected socket.

Since this method can be implemented on a much higher level compared to the methods presented in Sections 6.1.1 and 6.1.2, the actual string that the application uses to represent the remote host is visible, which, in most times, is a hostname rather than an IP address. Therefore, the offloaded socket connector is responsible for resolving the hostname to an IP address, so it's not necessary for the application to be able to contact a DNS server. Also, this allows the policy to be based on hostnames rather than only IP address.

## 6.2 Comparison of the potential solutions

To select which of the solutions will be investigated in the remainder of this chapter, a set of properties has been compiled. The following tables compares the three

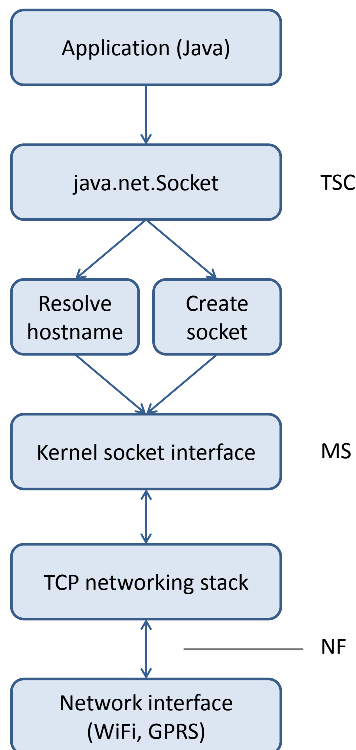



Figure 6.1: Policy enforcement moments. When an application requests a new connection, it uses the `java.net.Socket` class. The trusted socket creator (TSC) solution modifies the implementation at this place. The Java libraries create a socket in the kernel to resolve the hostname to an IP address and create another socket that is used for the outgoing connection. Both steps use the kernel socket interface, which will enforce a policy when modifying the `socket` system call (MS). Eventually, the TCP protocol stack transmits and receives IP packets to/from a networking interface. The netfilter (NF) subsystem is located at the connection between those two components.

solutions using this set of properties, the subsections below give a discussion for each of these properties. In the table, column MS refers to the modified `socket` system call (see Section 6.1.1), column NF refers to the netfilter solution (see Section 6.1.2) and column TSC refers to the trusted socket creator mechanism (see Section 6.1.3). Additionally, Figure 6.1 visually presents the positions in the flow of establishing a network connection where the three solution would enforce a policy.

| | MS | NF | TSC |
|---|---|---|---|
| Differentiate connection type (eg. WiFi, GPRS) | ✓ | ✓ | ✓ |
| Use hostname in policy | | | ✓ |
| Target port-listening policy to specific app | ✓ | | ✓ |
| Can be integrated with Apex (see Section 5.4) | | | ✓ |
| Existing connections will follow real-time policy change | | ✓ | |
| Allows unchanged Java code to use sockets | ✓ | ✓ | ✓ |
| Allows unchanged native code to use sockets | ✓ | ✓ | |
| Covert communication through DNS can be prevented | | | ✓ |

### 6.2.1 Differentiate connection type

For all three solutions, the policy can differentiate between the currently used connection type by the device. The most naive implementation could use separated policies for the different connection types, such that the policy enforcement part will switch between active policies when the network connection switches.

The `socket` system call can determine the currently used connection type by determining over what network interface the default gateway is reachable to select the currently active network policy. However, it may be impossible to determine whether the device is currently using the home network of the mobile operator or is roaming using this method. The netfilter solution can differentiate the outgoing network interface for each individual packet, a functionality which is already included in the subsystem and is often used in Linux-based personal and corporate firewalls.

Using the trusted socket creator mechanism, the trusted application can query the Android system for the currently used connection type at the moment a request for a new connection is being processed. Since this query is handled at higher levels, the system can differentiate between a device using the home network of the mobile operator or the network of a foreignoperator, allowing a higher level of granularity for policy decisions.

### 6.2.2 Use hostname in policy

Both the modified `socket` system call solution and the netfilter solution can only handle policies that are based on IP addresses. This is true for the modified `socket` system call solutions, because the `socket` system call is passed an IP address, which has been resolved by the application before requesting a network connection to be established. This is also true for the netfilter solution, because it operates on IP packets that have already been constructed by the operating system and inserted into the networking subsystem. Please note that IP packets only contain IP addresses for the source and destination fields and are unaware of the concept of hostnames.

The trusted socket creator mechanism is functioning on a higher level. Java applications normally request a connection to a hostname and the Java `java.net.Socket` class resolves the hostname to an IP address before it request the operating system to establish a connection. Therefore, implementing a policy enforcement on this level allows the policy to be based on hostnames, rather than IP addresses.

### 6.2.3 Target port-listening policy

To enforce a port-listening policy using the modified `socket` system call solution, the `listen` system call must be modified to enforce a policy that determines what ports an application is allowed to listen on. Among the information available to the kernel during a system call is the user ID of the process that performed the call, which the kernel can resolve to an individual application or a set of applications that have requested a shared user ID. Therefore, separate policies can be enforced for individual applications.

Using the netfilter solution, there is no way to prevent an application to initiate a listening operation on any port number. The netfilter subsystem is only invoked at the time a packet is received from a remote host that indicates a new connection is requested. At that time, a policy can be enforced that allows or disallows certain remote hosts to communicate with the system or the specific port number, but this policy is system-wide and cannot be targeted to a specific application.

Since a trusted socket creator solution has knowledge of which application requested a listening socket and what port number has been requested, it can target a port-listening policy to a specific application.

### 6.2.4 Integration with Apex

Apex (see Section 5.4) extends the mechanism used by the Android platform to decide whether a requested action is authorized. It does so by extending the ap-

propriate methods in the `PackageManager`, much like the trusted socket creator mechanism is implemented. The main difference is that the trusted socket creator adds additional methods to the `PackageManager`, while Apex modifies existing ones. The modifications made by Apex can also be performed on the methods that have been introduced for the trusted socket creator, such that a modified version of the policy engine used by Apex can be used to implement extended policies for the trusted socket creator mechanism.

To implement the two other solutions, changes have to be made to very different subsystems of Linux. Therefore, integrating it with Apex will not be possible, or at least be much more difficult compared to doing the integration of Apex with the trusted socket creator mechanism.

### 6.2.5 Real-time policy change

The proof of concept implementation of the trusted socket creator mechanism uses a network policy that is defined in the manifest file of an application and is therefore static. However, the policy can also be created by the user of the device, instead of an application developer, giving the user much more control over the network connections that application on his device can create. When this power is placed in the hands of the user, he can change the policy of application while they are running.

When using the modified `socket` system call solution or the trusted socket creator solution, any connection that has been established by an application will not be affected when the user changes the network policy that applies to that application. This is the case, since in both solutions the network policy is only checked at the time a new connection is created.

Using the netfilter solution, each and every individual packet is passed through the packet filtering subsystem of the Linux kernal and is tested against a chain of rules. When the chain of rules is modified, it will affect packets that are transmitted in the context of an existing packets. As a result, this solution allows the user to immediately shut down an unwanted network connection without requiring the application to be closed and restarted.

With additional effort, it would be possible to implement this behaviour when using a modified `socket` system call or when using the trusted socket creator. These efforts would require the system to keep track of open sockets and to which remote systems they are connected and require the system to iterate over the set of open sockets when the user changes the policy to verify whether any open connections are disallowed by the new policy. The connection can in turn be shut down using the `shutdown` system call, disallowing the application to use the open file descriptor any longer for communicating with the remote endpoint.

Alternatively, when using a trusted socket creator mechanism, it is possible to not return the file descriptor to a socket to the requesting application, but instead return a reference to an object that can be used for communication in the same way the `getListeningSocket` method returns an object that can be used to accept incoming connections. This way, the application will never hold a file descriptor to the socket and the `PackageManager` is therefore in the position to close existing connection when the policy is changed. However, this alternative will degrade performance of data transfers, since use of the interprocess communication mechanism would be required for all communication that is performed on sockets created in this way.

### 6.2.6 Required code changes

Except for disallowed connections, a modified `socket` system call behaves exactly like the original one, so therefore no code need to be changed in Java or native applications that create sockets. The same can be said for the netfilter solution, since the only difference will be that packets that are sent to disallowed destinations are dropped or rejected with an error message.

This is different for the solution using the trusted socket creator. Since the Android version of the `socket` system call will always fail when called by application that don't possess the `INTERNET` permission, native applications trying to create sockets will always get an error message when they use this system call. Instead, native application must be rewritten, such that they use the intercomponent communication framework to request a connected socket from the `PackageManager`. For application written in Java, this isn't an issue, since the Java libraries have been extended, such that they behave in exactly the same way as the original Android version, except when a requested connection is not allowed.

Most native application use the C library to create sockets, instead of directly calling the system calls. It is possible to modify the C library, such that the calls are intercepted and forwarded to the `PackageManager`, but such implementation should somehow relate functions calls that are used to translate a hostname into an IP address to function calls that are used to establish a network connection. This will be a complex task and prone to programming mistakes.

### 6.2.7 Preventing covert DNS channels

In 2008, Van Leijenhorst, et al. researched the viability of using DNS as a transport to covertly transfer information out of a host into the Internet or the other way around [28]. When an application that doesn't possess the `INTERNET` permission is capable of using DNS, this must not open up a covert channel that can be used by malicious applications.

When using the modified `socket` system call solution or the netfilter solution, application are responsible for resolving a hostname into an IP address, before being able to initiate a network connection. Therefore, the policy that is enforced when using one of those methods must allow communication with the DNS server, opening up the ability to abuse this channel for covert communication.

Alternatively, the C library could be modified, such that application that use functions in this library to resolve hostnames automatically use a trusted service in the system. Although this opens up the same channel, it might became less practical, since application will no longer be able to take complete control over the packets that are sent towards the DNS server.

The trusted socket creator solution proposed in the previous chapter effectively prevents such covert channel. The network policy is defined in terms of hostnames, rather than IP addresses, and the application isn't responsible for resolving hostnames into IP addresses, which is done by the `PackageManager`. The latter property implies that no DNS server needs to be allowed by the network policy.

Since the network policy is based on hostnames, the `PackageManager` will block outgoing connection requests based on hostname, even before DNS will be used to resolve such a hostname into an IP address. Therefore an application is unable to use DNS as a covert channel, unless a wildcard allow line is used (e.g. `*.dnstunnel.com`) that would open up communication to such hosts anyway.

### 6.2.8 Selecting a solution

Overall, the trusted socket creator solution has the most checkmarks, with the listed disadvantages to be that it may not be able to propagate a changed policy onto already established connections that are no longer allowed under the new policy and that native code that uses sockets needs to be modified to be functional.

Since most applications are written in Java and native code is mainly used in Android application for performance critical sections of an application this may not be a problem. Due to the latency and bandwidth limitation of any network connection compared to the computation speed of the central processing unit, network communication will never be part of the performance critical sections of an application. Even if a developer decides this is the case, he is still able to construct sockets in native code, but must use the interprocess communication framework. After a file descriptor to a kernel socket has been obtained, the actual communication can still take place in exactly the same way as is currently done in native code.

Because of these reasons, the trusted socket creator mechanism has been selected as the solution to research and implement in a proof of concept. The re-

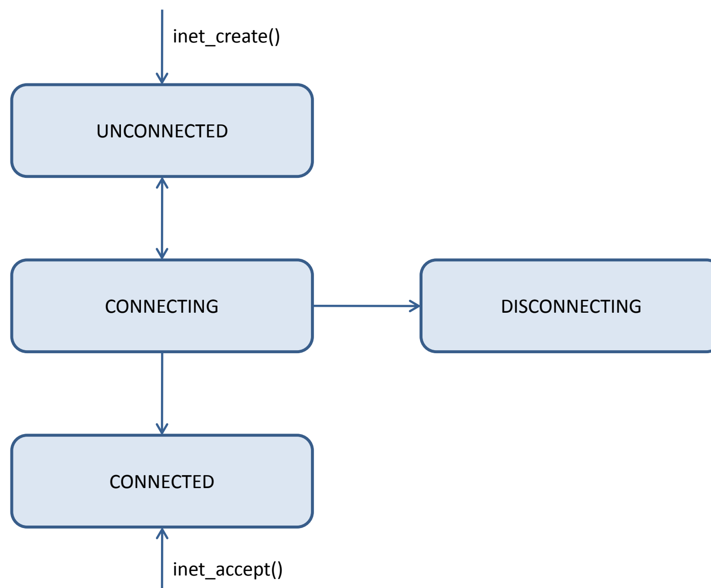



Figure 6.2: Linux socket state machine. Sockets that are constructed via the `socket` system call will initially be in the `UNCONNECTED` state; socket that are constructed via the `accept` system call will initially be in the `CONNECTED` state. Notice that no transition is possible when one of the `CONNECTED` or `DISCONNECTING` states are reached.

mainder of this chapter will therefore focus on this solution.

## 6.3 Socket state machine

Sockets in the `AF_INET` and `AF_INET6` address families can be created in two ways: either by calling the `socket` system call to allocate a new, unconnected socket in the kernel or by calling the `accept` system call to accept an incoming connection request and allocate a socket that represents this new connection. The information about the socket state machine used by the Linux kernel is based on the source code in the files in the `/common/net/ipv4/` directory of the Linux kernel source tree.

### 6.3.1 Outgoing TCP connections

After a socket has been allocated using the `socket` system call, it is placed in the `UNCONNECTED` state, as depicted in the state machine in Figure 6.2. The `connect` system call performs a transition into the `CONNECTING` state. When in this state, the connection request has been sent to the remote host, but no acknowledgment has been received by the system.

The socket remains in this state until the system receives the acknowledgment and transitions the socket into the CONNECTED state, the connection request is canceled by the application, receives an error message – e.g. the system is unreachable or no process is listening on the requested port – or the connection attempt times out. In those latter two cases, the socket is transitioned back into the UNCONNECTED state. When the connection request is canceled by the application, the socket is transitioned into the DISCONNECTING state. Note that when the socket is in the CONNECTED state and is closed by the application, the socket is deallocated and will therefore not transition into the DISCONNECTED state. However, the network stack will keep some state in memory to properly inform the remote host of this action.

### 6.3.2 Incoming TCP connections

When an allocated socket is used to accept incoming connections, the `listen` system call does not transition the socket to a different state. In fact, a socket that is listening for incoming connections can be reused to initiate an outgoing connection using the `connect` system call after using the `shutdown` system call to cancel the listening operation. As soon as a connection request is received and accepted by the application using the `accept` system call, a new socket is allocated in the CONNECTED state.

Currently, the INTERNET permission is enforced by the `socket` system call. Since allocating a new socket using the `accept` system call requires an existing socket that must have been constructed using the `socket` system call, `accept` doesn't need to enforce the INTERNET permission. As a result, the only way for a process that doesn't possess the INTERNET permission to obtain a socket is through the inter-component communication methods that have been described in Section 3.1.

### 6.3.3 After the CONNECTED state

Once a socket has reached the CONNECTED state – either by finishing a `connect` operation or by being allocated by the `accept` system call – it is not able to transition into another state.

Using the `close` system call, the application can inform the operating system that the connection to the remote system should be terminated and the socket in the kernel can be deallocated. Deallocating the socket means that the entire state is removed from the memory and therefore no state transition happens after the CONNECTED state.

A file descriptor to a socket that currently is in the CONNECTED state can be used to send data to and receive data from the remote endpoint and can be used to

change certain options on the socket that govern behavior and performance characteristics of this connection. System calls list `bind`, `listen` and `connect` return an error when they are passed a file descriptor to a socket in the `CONNECTED` state.

## 6.4 Implementation details

This section describes the modifications that has been done to the Android source code in order to implement a proof of concept for the policy enforcement method presented in Section 6.1.3. The components depicted in Figure 6.3 displays the components that have been modified and are involved in the creation of socket objects using the trusted socket creator concept.

In Java, the networking API is located in the `java.net` package. The most important classes in this respect are the `java.net.Socket` and `java.net.ServerSocket` classes. The implementation of the `Socket` class is modular, allowing different socket providers to provide an implementation. By default, an implementation provided by the Apache Harmony project [26] is used that uses JNI to eventually perform system calls like `socket` and `connect`. Hostname resolution is performed by the `java.net.InetAddress` class hierarchy, which uses functions in the C library to translate between hostnames and IP addresses. The C library uses the socket system calls to handle these requests.

The modular socket design allows a `SocketImplFactory` to be registered, which is invoked when a new `Socket` is instantiated. This results in a `SocketImpl` object, that is references by the `Socket` object and which will perform the actual socket operations. For the proof of concept of the trusted socket connector principle, the `SocketImpl` class has been extended, such that the methods involved with creating and connecting new sockets are overridden with an implementation that outsources these operations to the `PackageManager`.

### 6.4.1 Custom `SocketImpl` implementation

The custom `SocketImpl` – named `PMSocketImpl` to reflect the fact that operations are outsourced to the `PackageManager` – extends the existing `PlainSocketImpl` class that is part of the Apache Harmony implementation. This design allows the reuse of the code in the `PlainSocketImpl` class, by only overriding a small subset of the methods: `bind`, `connect`, `create`, `listen` and `accept`.

### 6.4.2 Outgoing TCP connections

When a new socket is created and connected to a remote host, the `create`, `bind` and `connect` methods are invoked on the `SocketImpl` object. First, the `create`

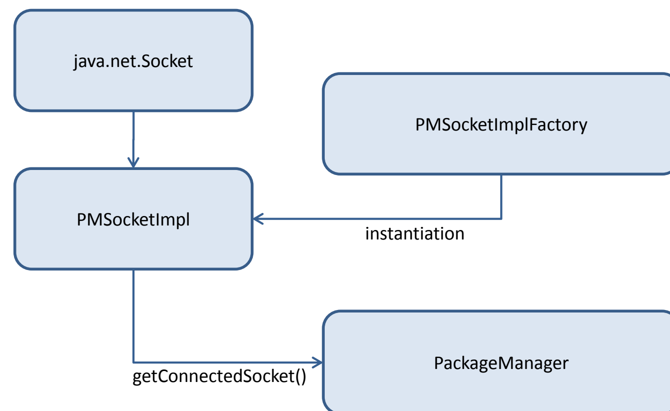



Figure 6.3: Socket classes involved in obtaining a file descriptor to a socket in the kernel. When a new `Socket` class is instantiated by an application, it requests an implementation class to be instantiated by the `PMSocketImplFactor`. When the application requests a socket to be created and connect to a remote host, it invokes methods on the `Socket` object, which delegates those tasks to the `PMSocketImpl` implementation. The `PMSocketImpl` class invokes `getConnectedSocket` on the `PackageManager`, such that a socket is created in the kernel and a connection request is initiated.

method is invoked, which normally results in the `socket` system call. The file descriptor of the new socket is stored in a member field of the implementation class. Second, the `bind` method is invoked to bind the socket to a local address. Finally, the `connect` method is invoked and the address of the remote host to connect to is passed as a parameter.

Regarding outgoing TCP connections, the `create` and `bind` methods in the `PMSocketImpl` class are effectively a no-operation method, since it doesn't perform any system call or intercomponent communication. These methods are overridden to prevent the default implementation in the `PlainSocketImpl` class to be executed, which would result in an exception to be thrown because the `INTERNET` permission is missing and therefore use of the `socket` system call is blocked.

The `connect` method uses the intercomponent communication facilities available in Android – which have been described in Section 3.1 – to invoke the `getConnectedSocket` method on the `PackageManager`. The `PackageManager` is part of the Android framework and has been extended with this method as part of this thesis. The `getConnectedSocket` method is therefore executed in the `system_server` process, which possesses the `INTERNET` permissions that is required to actually create a socket in the kernel.

The getConnectedSocket combines the operations of resolving the hostname to an IP address, creating a socket and connecting it to the requested remote host. Only when the socket has been fully connected – and therefore the state machine in the kernel is in the CONNECTED state – the file descriptor of the socket is returned back to the calling application. Any problem that occurs when resolving the hostname, creating the socket or connecting to the remote host is passed back to the application.

Next, the PMSocketImpl parses the response from the getConnectedSocket invocation. When a problem has occurred, the error code is used to construct the appropriate exception – e.g. UnknownHostException or SocketTimeoutException – and a human-readable message is included, such that the behavior of this implementation is identical when compared to the default implementation provided by Apache Harmony. When no problem has occurred, the file descriptor is stored in the appropriate member field, such that the methods that are implemented in the PlainSocketImpl superclass will use this file descriptor and act as expected.

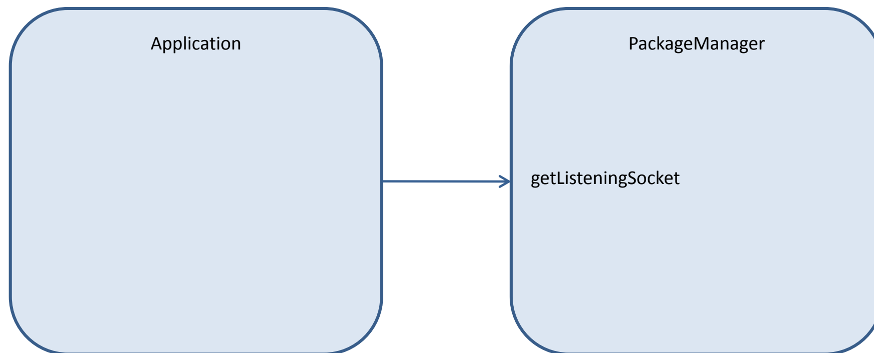
### 6.4.3 Incoming TCP connections

For incoming TCP connections, an application in Java uses the ServerSocket class to set up a socket in the kernel that is registered to a specific TCP port number to listen for incoming connection requests. When a ServerSocket object is instantiated, it obtains a SocketImpl class and invokes the create, bind and listen methods to set up a socket and put it into a listening mode.

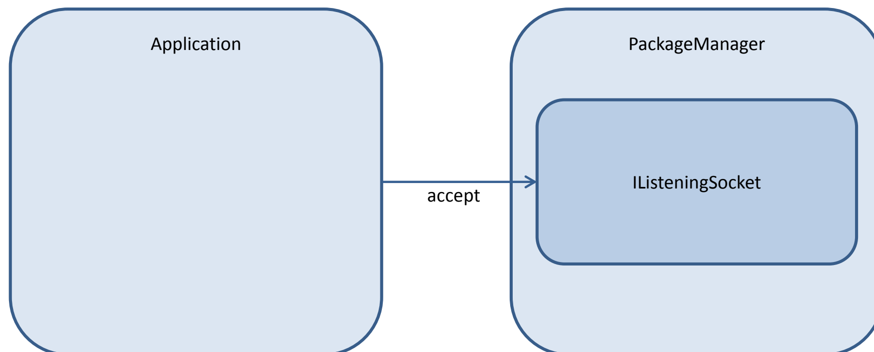
After successfully putting a socket into listening mode, an application can obtain a connected socket using the accept system call. This system call first polls a queue of waiting connection requests and returns a file descriptor to the first socket that is waiting. If no incoming connection request is waiting, the accept system call blocks and the thread of execution is put to sleep until a new connection request is received. At that point in time, the thread wakes up and returns a file descriptor to the new socket.

When a ServerSocket has attached to a TCP port number, the application can invoke the accept method to retrieve a Socket object that represents a connection to the remote endpoint that initiated the connection. The accept method invokes an equally named method on the referenced SocketImpl class, which is passed a reference to the new Socket object. The accept method of the SocketImpl class obtains the file descriptor from the accept system call and stores it in the Socket object that has been passed in.

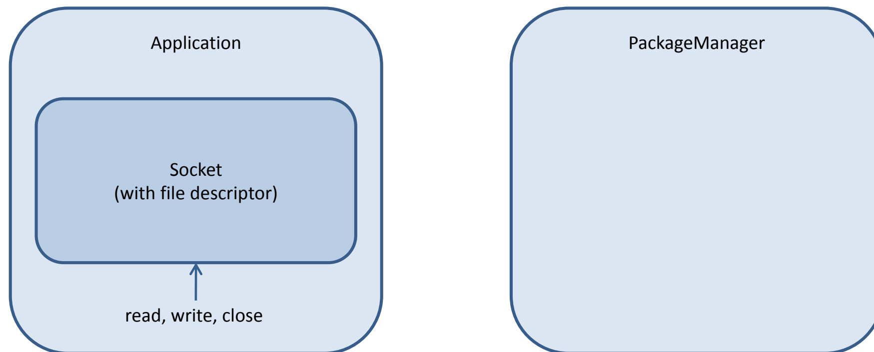
The custom PMSocketImpl overrides the bind, listen and accept methods. For setting up a server socket, the bind method just stores the local IP




(a) Application invokes `getListeningSocket` on the `PackageManager`, which returns a reference to an `IListeningSocket` object.




(b) Application invokes `accept` on the `IListeningSocket`, which returns the file descriptor to a socket connected to a remote host that initiated a connection.




(c) Application invokes data transfer operations directly on the file descriptor without using the Inter-Component Communication framework.

Figure 6.4: Application accepting an incoming connection. First, the application invokes `getListeningSocket` on the `PackageManager`, which allocates an `IListeningSocket` in the process of the `PackageManager` which holds a file descriptor to a socket that is placed into listening mode (a). The application can accept an incoming connection request by invoking `accept` on the `IListeningSocket` object (b). The application places the file descriptor in a `Socket` object, on which all required methods can be invoked locally (c).

address and port number that are passed, which indicate the local port that the application wants to listen on. When the `listen` method is invoked, the stored local endpoint identification is sent to the `getListeningSocket` method in the `PackageManager`. This methods verifies whether the requesting application is allowed to listen on the specified TCP port and sets up the listening socket if this is the case.

If the `getListeningSocket` method were to return a file descriptor that is put into listening mode, the application would be able to reuse this socket to connect to other hosts by directly using system calls and that way it would bypass the policy enforcement as described in Section 6.3.2. Instead an object that implements the `IListeningSocket` is returned to the application, which implements a `close` and a `accept` method. The application can invoke these methods through the `IBinder` framework.

The `PMSocketImpl` class stores the `IListeningSocket` reference that is returned by the `accept` method in a member field. When the socket is closed – e.g. the application invokes the `close` method of the `ServerSocket` class – the `close` method on the `IListeningSocket` class is invoked. Note that the application doesn't have a file descriptor that gives access to the listening socket, but has a reference to an object in the `PackageManager` that in turn controls a listening socket. The `PackageManager` is therefore in the position to enforce a security policy.

When the `accept` method is invoked on the `IListeningSocket` that has been instantiated by the `getListeningSocket` method, the `accept` system call is invoked. As soon as it returns, the IP address and port number of the remote endpoint are requested through the file descriptor to the new socket, such that it can be tested against a policy for incoming connections. If the identification matches any policy entry, the file descriptor to the new socket – which is in the `CONNECTED` state – is returned to the application. Otherwise, a `SecurityException` is returned.

As soon as the `accept` invocation returns, the `PMSocketImpl` class stores the retrieved file descriptor in a newly created `SocketImpl` object, which is returned to the application in the form of a `Socket` object. This `Socket` object can be used for communication with the endpoint that initiated the connection. The entire process is displayed in Figure 6.4.

### 6.4.4 Process termination

When an application ends and the (Linux) process terminates, all file descriptors that are held by this process are automatically closed. In effect, this means that sockets that are used by the application are closed, regardless of whether

the socket was used for an outgoing connection, an incoming connection or is currently in a listening mode waiting for new incoming connection requests.

When using the trusted socket creator mechanism, file descriptors to sockets are held in an external process with respect to the application process that uses them. When the `PackageManager` is used to create a socket for an outgoing connection, it will only hold the file descriptor for the period required to establish the connection and pass it back to the requesting application. After this has been done, the file descriptor held by the `PackageManager` is closed and therefore the requesting application is the only one with the file descriptor. When the application process terminates, the kernel will therefore automatically close the socket and tear down the TCP connection.

The same is true for incoming TCP connections as soon as they are accepted. However, the socket that is placed in listening mode and that is waiting for new incoming connection requests is never passed to the application process and therefore is only reachable from the process that hosts the `PackageManager`. When the application that initiated the listening operation is terminated, the socket will therefore not be closed automatically by the kernel.

As a result, the `PackageManager` must monitor the requesting process and explicitly close the listening socket at the moment the process terminates. The Android inter-component communication framework provides a notification mechanism that can be used to correctly handle the situation that arises when the process that hosts a referenced object terminates, such that the application that holds the reference can take appropriate action.

To use this mechanism, the `getListeningSocket` method accepts an additional argument that allows the requesting application to pass a reference to any `IBinder` object that resides in the process. The `PackageManager` registers a `DeathRecipient` object that will be invoked as soon as the application process terminates and the `IBinder` object will not be reachable anymore. The event handler will close the listening socket.

## 6.5 Socket policy

When the implementation explained in Section 6.4.1 is in place, the `getConnectedSocket` method is the entry point for application that don't possess the `INTERNET` permission to obtain a socket and be able to communicate with other systems. Therefore, this method is the location to implement the policy enforcement logic. For incoming connections, the `getListeningSocket` and `accept` methods are the entry points for setting up a listening socket and accepting incoming connections.

To specify what systems an application may connect to, a policy must be defined. The definition for the implemented policy has been inspired by the Java 2 `SocketPermission`[25] definition. A policy consists of zero or more "allow lines" that are included in the manifest file of the application. An application with no allow lines cannot access any host; each allow line grants access to an additional set of hosts or TCP ports.

The format of individual allow lines has been taken from the Java 2 `SocketPermission` policy format. Using EBNF[1] notation, an allow line is defined as:

```
allow-line = (hostname | domain-suffix | ip-address),
                [:portrange];
hostname = (name-part, '.')*, (name-part);
domain-suffix = '*.', hostname;
ip-address = ip4-address | ('[', ip6-address, ']');
ip4-address = ''IPv4 address in dotted notation'';
ip6-address = ''IPv6 address in RFC 1924 notation'';
portrange = portnumber, ['-', [portnumber]];
portnumber = ''TCP port number'';
```

Thus, such an allow line specifies the host or set of hosts that it applies to and optionally a port number or a range of port numbers. The host can be specified as either an IP address (e.g. 131.155.2.83), a hostname (e.g. "www.tue.nl") or a set of hosts that share a domain (e.g. "*.win.tue.nl").

Using this syntax, the following are examples of valid allow lines:

`mail.example.com:143` Grant access to the IMAP port (143) on `mail.example.com`

`10.0.0.24:6667-6670` Grant access to the ports 6667, 6668, 6669 and 6670 on the host with IP address `10.0.0.24`

`[ff02::fb]` Grant access to all TCP ports of the IPv6 mDNS address `ff02::fb`

`*.example.com:80` Grant access to the HTTP port (80) of all hosts that are in the `example.com` domain, e.g. www.example.com and `www.department.example.com`

`ftp.example.com:21` Grant access to the FTP port (21) on `ftp.example.com`

`ftp.example.com:1024-` Grant access to ports 1024 and above on `ftp.example.com`

[1] Extended Backus-Naur Form

The policy for incoming connections is two-fold: the first policy contains a list of port numbers that the application may listen on and the second policy contains a list of remote endpoints that are allowed to connect to the application. The remote endpoint policy resembles the one described for outgoing connections, except that using hostnames is not possible, as the remote endpoint is identified by an IP address.

### 6.5.1 Policy enforcement

To enforce the defined policy, the `getConnectedSocket`, `getListeningSocket` and `accept` methods must verify whether a connection or listening request is allowed according to the policy, before creating the socket and initiating the connection establishment. Therefore, the `PackageManager` has also been extended with some methods that can check the socket permission for a given application. The `checkConnectPermission` method iterates over the set of allow lines applicable to an application and determines whether or not the requested connection is allowed. The `checkAcceptPermission` does the same, but uses the set of incoming allow lines. Finally, the `checkListenPermission` method iterates over the set of allowed port numbers to determine whether a requested `getListeningSocket` invocation is allowed.

When the `checkConnectPermission` or `checkAcceptPermission` methods iterates over the set of allow lines, the first line that matches will cause the method to finish and return PERMISSION_GRANTED. However, when no allow line matches the hostname and port number of the resource the application wants to connect, the method will return PERMISSION_DENIED. When PERMISSION_DENIED is returned, the `getConnectedSocket` method will abort the request and return a `SecurityException` to the requesting application or the `accept` method will close the newly accepted connection and return a `SecurityException` to the requesting application.

The `checkListenPermission` iterates over the set of port numbers that are allowed for this application. When the port number that the application requested to listen on is present in this set, the method will return PERMISSION_GRANTED, causing `getListeningSocket` to proceed with creating a `ServerSocket` and placing it in listening mode on the requested port. When the requested port number is not present in the list of allowed port numbers, `checkListenPermission` returns PERMISSION_DENIED and a `SecurityException` is returned to the requesting application.

The result is that applications that don't have the INTERNET permission can still communicate with the Internet when more fine-grained permissions are specified. Outgoing connections can only be established to hosts and ports that are

allowed by the outgoing policy, while incoming connection are only possible to allowed port numbers and from hosts that are allowed by the policy.

## 6.6 Conclusion

This chapter proposed an enhancement of the Android permission model, such that a more fine-grained Internet policy can be enforced for individual applications. Three alternative solutions have been introduced: 1) one based on modifying the `socket` system call, 2) one based on the netfilter packet filtering subsystem of the Linux kernel and 3) one based on a trusted socket creator that enforces a network policy when establishing new connections. The remainder of the chapter presented a solution based on the trusted socket creator concept. It has been explained that TCP sockets cannot be reused for communication with other hosts after they have been connected to an endpoint. The result of this enhancement is that application developers can now request access to particular hosts and ports on the Internet, instead of requesting access to the entire Internet, solving the objections raised by Barrera, et al. in [3]. Consequently, this proposal incorporates the principle of least privilege into the Android platform regarding Internet access.

# 7

# Conclusion and future work

## 7.1 Proof of concept

A proof of concept of the trusted socket creator mechanism has been created by the author to demonstrate the feasibility of this solution as described in Sections 6.4 and 6.5.1. To test the modifications, an application that uses the `java.net.*` classes has been created that establishes a connection to a remote host and writes debug log messages when the connection has been established and when an exception has been caught. As expected, connections to hosts that are not allowed by the policy are denied and a `SecurityException` was thrown, while for all connection requests to hosts that are allowed by the policy, the behaviour didn't change with respect to the original functionality of the Android platform.

In the same way, the scenario of incoming connections has been tested by requesting a listening socket on a port number that is allowed by the policy and establishing a connection to this port number from a remote host. Requesting a listening socket on a port number that's not included in the policy is successfully rejected and results in a `SecurityException` to be thrown.

Most applications don't use the `java.net.*` classes directly, but rely on higher layer classes that implement protocol details. For example, opening an HTTP connection is usually performed using the `java.net.URL` classes that are part of the standard Java class hierarchy. These classes internally use the `java.net.*` classes to actually set up a network connection, but also implement the client role of an HTTP message exchange.

To test the functionality of the proof of concept implementation, an application has been created that has a single allow line "*.google.com:80", which allows the application to only connect to port 80 on any host that belongs to the google.com domain. Since the default behavior of Google is to redirect the user to a country-specific website like google.nl, the Java HTTP client automatically tries to connect to this host to retrieved the redirected document. However, this host is denied by the policy and therefore a `SecurityException` is thrown. The process is illustrated in Figure 7.1.

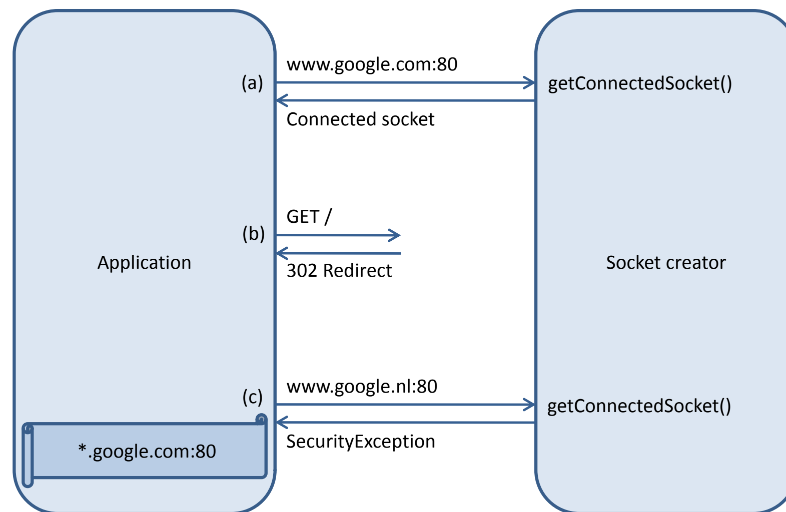



Figure 7.1: Application connecting to the Google homepage. The application requests a socket that is connected to a connection to www.google.com on port 80 (a) and requests the homepage of the website using an HTTP request (b). The webserver determines the country from which the request originated and redirects the client to a local version of the search engine, for example www.google.nl. Following the redirect, the client tries to open a connection to www.google.nl on port 80 (c). Since this endpoint doesn't match any allow line, the connection request is denied and a `SecurityException` is returned.

## 7.2 Conclusion

This thesis attempted to come to an answer to the following research question, as introduced in Section 1.1:

*How can the permission model employed by the Android smartphone platform be improved, such that some known weaknesses are fixed and the existing protection and usability are preserved?*

Following the results of the five subquestions – which are presented with an anser in the next subsection – and the work presented in Chapter 6 and discussed in this chapter, the following answer to the research question that serves as the topic of this thesis can be formulated:

*Using a trusted socket creator that is not under the control of the application that requests network communication, the Android platform can be extended to support a more fine-grained permission model for Internet access, which fixes the weakness that follows from the coarse-grained INTERNET permission without reducing the usability of the Android platform or any application.*

### 7.2.1 Subquestions

For each of the individual subquestions presented in Section 1.2, an answer has been formulated, which are presented in this subsection.

#### Q1. How is the current Android permission model designed?

Chapter 2 describes how the current permission model is designed in Android. Applications contain a manifest file that state a set of permissions they need to operate correctly. During installation of an application, the user can either grant access to those permissions and resume the installation or deny access and abort the installation of the application.

#### Q2. Which platform component(s) contribute to the implementation of the permission model?

A very important mechanism in enforcing the permission model is the intercomponent communication mechanism explained in Section 3.1, which allows application components to communicate with components that are hosted in different Linux processes. In this way, the `PackageManager` component – which is part of the Android platform and described in Section 3.2 – is hosted in the `system_process` process, which cannot be influenced by other applications. The `PackageManager` is the component that implements the decision-making part of the permission model, while individual services are responsible for querying the `PackageManager` to determine whether the requestion application has permission to perform the requested operation or access the requested resource.

#### Q3. What vulnerabilities are known to exist in the current model?

Several vulnerabilities have been identified in the current model, which are described in Chapter 4. One of the vulnerabilities that have been presented in Section 4.2.1 is the fact that 60% of the applications in the Android Market request the `INTERNET` permission, allowing them to communicate with any host on the Internet. Since a lot of application only request this permission to be able to retrieve advertisements from Internet hosts, research like [3] propose a fine-grained permission system for Internet access as a future enhancement to Android.

#### Q4. What improvements have already been developed?

Several improvements have been proposed, which have been summarized in Chapter 5. Three frameworks have been presented: Kirin (Section 5.2), SCanDroid (Section 5.3) and Apex (Section 5.4). Chapter 5 includes a discussion about these

frameworks and proposes advantages that can be gained by combining the frameworks.

#### Q5. How can the permission model be improved to fix the identified vulnerabilities?

Three solutions have been identified to implement a fine-grained policy for Internet access into the Android permission model, which have been described in Section 6.1. The remainder of Chapter 6 forms the contribution of this thesis and presents the concept of a trusted socket creator, which allows applications to obtain a file descriptor to a connected socket, such that the `PackageManager` is capable of enforcing a fine-grained network policy over outgoing and incoming network connections.

## 7.3 Future work

The solution presented in Chapter 6 has been scoped to only include TCP communication. In future research, this can be extended to include more types of communication and more expressive policies.

### 7.3.1 UDP

When using TCP, the kernel maintains a state machine that disallows a socket to move from the `CONNECTED` state into any other state. Therefore, TCP sockets that have been connected cannot be reused to communicate with another host. The concept of connecting to a remote host is also inherent in TCP, as it is designed to allow two-way communication between two hosts that keep a state to remain synchronized.

Since UDP is designed as a connectionless protocol, the concept of "connecting" to a remote host is not applicable to the protocol. To use UDP sockets in Linux, an application uses the same socket API that is used for TCP. This means the `connect` system call is available and has actually been implemented for UDP. After using this system call, all sending operations use this destination address by default and all receiving operations discard data coming from different source addresses. This is mainly used to ease application programming when using UDP to communicate with a single remote host.

While the `connect` system call mimics the behaviour of a TCP connect for UDP, no state machine is maintained by the kernel to enforce proper connection control. As a result, it is possible to call `connect` multiple times on the same socket, each time changing the remote address stored by the kernel. Effectively, this allows an application that has a file descriptor to a UDP socket to communicate with any

UDP host that is reachable. Therefore, to enforce a network policy, an application must never receive a file descriptor to a UDP socket.

### 7.3.2 Extended policies

The kind of policies that are implemented for this thesis are rather fixed and are created by the application developer. For some applications this may be sufficient, i.e. applications that know a priori to what hosts they are going to connect. While a Facebook or Twitter application falls in this category, applications like generic mail clients that allow the user to configure the mail server to use are unable to benefit from such extension.

It may therefore be a nice extension to allow a user to define the policy. For example, a mail client application may contain a settings activity which is used to configure the mail server to connect to. When the mail server setting is changed, the application can invoke a system activity that asks user consent for connecting to the specified host and modifies the network policy for the application accordingly. Another possible way to do this is to ask the user to make a decision for any connection request that isn't matched by the current policy.

When socket creation is outsourced to the system server, more dynamic policies can be implemented in exactly the way it has been done by Apex (see Section 5.4). This way, access to certain hosts can be based on other properties like time of day or limited to a certain number of allowed connections per day.

An extended policy may also take the currenlty available connection types into account, which may especially be useful when the user has the capability to influence the policy. As an example, a user may want to create a policy that allows an application to contact specific servers when connected to the Internet using either a WiFi connection or the mobile network of the operator, while disallowing the access to those same servers when roaming on the mobile network of a foreign operator.

# Bibliography

[1] About.com cell phones. What Is a Smartphone? http://cellphones.about.com/od/glossary/g/smart_defined.htm. Retrieved on 2011-02-03.

[2] Android, Inc. Android Developers Guide. http://developer.android.com/guide/index.html, 2011. Retrieved on 2011-03-08s.

[3] D. Barrera, HG Kayacik, P.C. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 73–84. ACM, 2010.

[4] J. Burns. Mobile application security on android. *Black Hat'09*, 2009.

[5] A. Chaudhuri. Language-based security on Android. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, pages 1–7. ACM, 2009.

[6] Dalvikvm.com. Dalvik Virtual Machine insights. http://www.dalvikvm.com/. Retrieved on 2011-02-03.

[7] Ben Elgin. Google Buys Android for Its Mobile Arsenal. *Business Week*, August 2005.

[8] W. Enck, M. Ongtang, and P. McDaniel. Mitigating Android software misuse before it happens. Technical report, Technical Report NAS-TR-0094-2008, Pennsylvania State University, 2008.

[9] W. Enck, M. Ongtang, and P. McDaniel. Understanding android security. *Security & Privacy, IEEE*, 7(1):50–57, 2009.

[10] Seth Fogie. BlackBerry Firewall. http://www.informit.com/guides/content.aspx?g=security&seqNum=348, 2009. Retrieved on 2011-06-15.

[11] Jay Freeman. Mobilesubstrate. http://iphonedevwiki.net/index.php/MobileSubstrate. Retrieved on 2011-06-16.

[12] A.P. Fuchs, A. Chaudhuri, and J.S. Foster. SCanDroid: Automated Security Certification of Android Applications. In *Submitted to IEEE S&P'10: Proceedings of the 31st IEEE Symposium on Security and Privacy*. Citeseer, 2010.

[13] GSM Association. History. http://www.gsmworld.com/about-us/history.htm. Retrieved on 2011-02-03.

[14] iPhoneHeat. Firewall ip for iphone goes 1.2. http://www.iphoneheat.com/2009/12/firewall-ip-for-iphone-goes-1-2/. Retrieved on 2011-06-20.

[15] C. King. *Advanced BlackBerry Development*. Apress Series. Apress, 2010.

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

[17] PalmSource, Inc. OpenBinder version 1.0. http://www.angryredplanet.com/~hackbod/openbinder/docs/html/index.html. Retrieved on 2011-02-18.

[18] PCMag. Definition of: Smartphone. http://www.pcmag.com/encyclopedia_term/0,2542,t=Smartphone&i=51537,00.asp. Retrieved on 2011-02-02.

[19] Ruediger Rill. Firewall ip. http://yllier.webs.com/firewall.html. Retrieved on 2011-06-16.

[20] Rodrigo. Droidwall - android firewall. http://code.google.com/p/droidwall/. Retrieved on 2011-06-20.

[21] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X.F. Wang. Soundminer: A Stealthy and Context-Aware Sound Trojan for Smartphones.

[22] W. Shin, S. Kiyomoto, K. Fukushima, and T. Tanaka. Towards Formal Analysis of the Permission-Based Security Model for Android. In *Wireless and Mobile Communications, 2009. ICWMC'09. Fifth International Conference on*, pages 87–92. IEEE, 2009.

[23] W. Shin, S. Kiyomoto, K. Fukushima, and T. Tanaka. A Formal Model to Analyze the Permission Authorization and Enforcement in the Android Framework. In *International Symposium on Secure Computing (SecureCom-10)(to appear)*, 2010.

[24] W. Shin, S. Kwak, S. Kiyomoto, K. Fukushima, and T. Tanaka. A small but non-negligible flaw in the Android permission scheme. In *IEEE International Symposium on Policies for Distributed Systems and Network*, 2010.

[25] Sun Microsystems, Inc. Permissions in the Java(TM) 2 SDK. http://download.oracle.com/javase/1.4.2/docs/guide/security/permissions.html. Retrieved on 2011-05-10.

[26] The Apache Software Foundation. Apache Harmony. http://harmony.apache.org/index.html. Retrieved on 2011-05-02.

[27] UNIX Manual Page. chmod (2).

[28] T. van Leijenhorst, K.W. Chin, and D. Lowe. On the Viability and Performance of DNS Tunneling. ICITA, 2008.

[29] Wikipedia. Smartphone. http://en.wikipedia.org/wiki/Smartphone, 2011. Retrieved on 2011-02-03.

# Glossary

**Marshalling** Transforming a structure of connected objects, e.g. in a Java environment, into a stream of bytes or characters, that can be transmitted over a communication channel.

**Unmarshalling** The reverse operation of *marshalling*, i.e. transforming a stream of bytes or characters into a structure of linked objects.

**Virtual Machine** A software implementation of a certain instruction set, allowing application encoded in that instruction set to be executed on hardware that uses a different instruction set. A virtual machine can also act like a logical barrier between code running inside and code running outside the virtual machine.

# Index

AccessManager, 42
activity, 11
aidl, 22
Android Runtime, 10
Android, Inc., 1
Apache Harmony, 59
Apex, 41, 53

Binder, 10, 21
  onTransact(), 22
BinderProxy, 21
broadcast receiver, 11

CEPT, 1
content provider, 12
Context, 23
covert communication, 26

Dalvik Virtual Machine, 10, 13
DeathRecipient, 64
dx, 13

Google, Inc., 1
GPRS, 1
GSM, 1

IBinder, 21
  transact(), 21
ICC, 13, 19
IListeningSocket, 63

Java Native Interface, 21

Kirin, 34

manifest file, 29, 40
MediaRecorder, 28

netfilter, 49

Open Handset Alliance, 1
OpenBinder, 13, 19

PackageManager, 23, 42, 60
  checkUidPermission(), 23
  getConnectedSocket(), 60
  getListeningSocket(), 63
Parcel, 21
permission
  ACCESS_COARSE_LOCATION, 27
  ACCESS_FINE_LOCATION, 27
  BLUETOOTH, 27
  CALL_PHONE, 25, 26
  INTERNET, 14, 26–28, 33, 34, 50, 58, 60
  MODIFY_AUDIO_SETTINGS, 26
  READ_PHONE_STATE, 31
  READ_SMS, 28
  RECEIVE_BOOT_COMPLETED, 31
  RECEIVE_SMS, 31
  RECORD_AUDIO, 28, 33
  SEND_SMS, 14, 23
  WRITE_APN_SETTINGS, 26
  WRITE_SETTINGS, 26

WRITE_SMS, 28
PlainSocketImpl, 59
PMSocketImpl, 59
PolicyResolver, 42
Prolog, 34

sandbox, 12
SCanDroid, 37
ServerSocket, 59
service, 11
smartphone, 1
SmsManager, 23
Socket, 59
system_process, 23