SAARLAND UNIVERSITY
COMPUTER SCIENCE

Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science
Master's Program in Computer Science

Master's Thesis

User-controlled Internet Connections in Android

submitted by

Sven Obser

on December 22, 2011

Supervisor
Prof. Dr. Michael Backes

Advisor
Sebastian Gerling

Reviewers
Prof. Dr. Michael Backes
Dr. Matteo Maffei

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, December 22, 2011

Acknowledgments

I would like to thank Prof. Dr. Michael Backes for the most inspiring and most exciting lectures in my study and for introducing me into the world of cryptography and information security. Thanks a lot for supervising this thesis. I would also like to express my gratitude to Dr. Matteo Maffei for agreeing to review the thesis.

Special thanks to my advisor Sebastian Gerling for offering me such an interesting topic, for his support throughout the creation of this work, and for his tremendous feedback to improve several parts of this thesis.

Lena, I thank you for your love and support in the last years! No matter how bad my mood is, you always manage to make me laugh.

The last words I devote to those who supported me since the beginning of my life. I want to thank my father and my mother for building up a live as we had and for their continuous and unlimited support. This work would not have been possible without your great support.

Abstract

Modern mobile devices (e. g. smartphones and tablets) hold more and more sensitive personal data and are additionally able to execute untrustworthy code from third-party developers. To protect users from malicious applications, the security model of Android is based on mandatory access control. For every dangerous operation, the developer of an application has to explicitly request a corresponding permission (e. g. to read contacts or to access the Internet). As soon as an application gets installed, the user has to approve the requested permissions. One permission that is requested by many applications is the Internet permission.

Unfortunately, the current security model is quite coarse with respect to this Internet permission. Neither the user nor the developer is able to restrict the network access of applications in a fine-grained manner. Developers can only request full Internet access and the user can either grant this permission and install the application or reject the permission by not installing the application at all.

In this thesis, we present a refined permission model for Android that supports fine-grained Internet connections. Further, we have modified the stock Android firmware and implemented *user-controlled Internet connections in Android* that allow to configure the Internet permission of each application dynamically. We measured the network performance of our solution on a Nexus One developer phone and found that the average throughput does not decrease significantly. This shows that our contribution can practically increase the security of Android by giving users more control over established Internet connections on their phones.

Contents

1 Introduction

Since the first release of Apple's iPhone in 2007, the market for high-end mobile devices (e. g. smartphone and tablet devices) is rapidly growing. Google released *Android*, its competitor to Apple's mobile operating system *iOS*, in November 2007. Next to iOS (Apple), Symbian (Nokia), and the BlackBerry OS (Research In Motion), Android has become one of the most important mobile operating systems for smartphone devices [32].

All these mobile platforms have in common is the functionality of the mobile device is highly customizable by the user. The functionality can be enhanced by the installation of third-party applications. In general, these applications are available in a dedicated application store of the platform (e. g. the Apple App Store or the Android Market). These markets offer a large variety of free and non-free applications and allow the user to easily install free or purchase non-free applications.

However, the extensibility of the mobile devices also has some negative impact on the data security of the mobile devices. The growth of the mobile market made smartphones attractive targets for attackers. We see the same threats arising on these platforms we already know from personal computers. Thereby, malware is one of the biggest, if not even the biggest, security threat for these mobile platforms. Banking Trojans like *Zeus* also target Android in order to steal the one-time password of a transaction, the mobile transaction authentication number (mTAN) [37]. This mTAN is sent to the user's mobile phone and gets forwarded automatically by the installed Trojan to the attacker's server. Together with the login credentials of the user, the mTAN allows to conduct transactions on behalf of a user.

Android tries to solve this security threat with an extensive security model. A sandboxing approach and other security techniques (e. g. stack protection) are used to limit the impact of malicious applications. Further, a permission system that is based on mandatory access control limits the access to security critical functionalities and resources on the device. The user grants these functionalities whenever installing a new

application, such that the new application is, for example, able to access the Internet or to read the user's address book.

Unfortunately, Android's permission system relies heavily on the user's abilities to estimate the risk of permission requests. However, the average user is not always able to decide whether to install the application or not. The user's only intention is to get the promised functionality of the application; however, the threats that arise from the granted permissions are, in general, quite complex and require profound understanding of information security. For example, granting the application the permission to read the contact list is not dangerous in general, but in conjunction with the Internet permission, an attacker is able to steal this information and transmit it to a remote server.

Especially the Internet permission is important for an attacker to deliver private information from the user's mobile phone to the attacker. We think that the current design of Android's Internet permission is too coarse, so that users are not able to decide whether granting the Internet permission is secure or whether it leads to a security threat. We shall try to illustrate our opinion with a fictional example.

1.1 Motivating example

Assuming the average user Alice is browsing the Android Market looking for the mobile banking application of the *West Pacific Bank*. The only available application in the Android Market is the *West Pacific Mobile Banking* application provided by a developer called *West Pacific Bank*. The provided description and the screenshots of the application look unsuspicious. However, the application has just been uploaded and does therefore not have any user reviews.

Alice decides to install the application; however, Alice first checks the requested permissions of the application in detail, which reveals that the application requires *full Internet access*. This permission request seems to be absolutely plausible for a mobile banking application, so that Alice finally installs the application. After starting the application and entering the login credentials of the bank, the application states that Alice's device is unfortunately not supported. In fact, the *West Pacific Mobile Banking* application was a Trojan that passed the login credentials of the user to a remote server. Although Alice tried to be as careful as possible, she was not able to detect the malicious behavior without analyzing the application in detail.

We think that a refinement of Android's Internet permission can help to defend against this kind of security threat. In our example, this would mean that an official banking application would not request *full* Internet access but rather *restricted* Internet access to `banking.westpacific.com`. If we assume that it is good practice to request fine-grained Internet permission, it would be suspicious if a banking application requests full Internet access or if it wants to access a suspicious server.

1.2 Contribution

Our contribution to Android is a refined permission model that allows application developers to request fine-grained Internet permissions. Additionally, we give the user more control over the Internet connections that are established by the installed Android applications. Instead of granting applications full Internet access, the user decides to which destinations an application can finally establish a connection.

In addition to the redefined permission model, we have provided an implementation of *user-controlled Internet connections in Android (Ucicia)* and have modified the stack Android firmware in order to enforce the redefined permission model.

1.3 Outline

We structured the rest of this thesis as follows: Firstly, we introduce Android, Google's mobile operating system in Chapter 2. This includes an overview of Android's version history since its first official release and a detailed consideration of the operating system's internals. Further, we describe how new Android applications can be developed and how they are deployed to the end-user. In Chapter 3, we show the current Android attacker model and describe the different techniques that are deployed in Android's security model to protect the device against the existing security threats.

Chapter 4 introduces the basics of the underlying Internet communication and presents access control mechanisms that are used to enforce a permission system. Further, we provide a first high-level overview of our contribution. In Chapter 5, we define the modified permission model that supports fine-grained Internet connections and additionally present the key functionalities of our contribution. In Chapter 6, we provide a detailed view on how we modified the Android firmware in order to implement the refined permission model and measure the impact of our modifications on the performance in Chapter 7.

2 The Android operating system

Android is an open source operating system for mobile devices (e. g. smartphones and tablets) developed by the Open Handset Alliance. The main contributor to the project is the software company *Google*. Thus, Android is often referred to as Google's smartphone operating system. Since the first release of Android in November 2007, it has become one of the most important operating systems for high-end mobile devices [30, 31].

Like other smartphone operating systems (e. g. Apple's iOS or Nokia's Symbian), Android allows to extend the functionality of the device by installing third-party applications (so called *apps*). For this purpose, Android consists of a full software stack that utilizes the capabilities of modern mobile devices and provides an application programming interface (API) that allows application developers to easily access the capabilities of these mobile devices.

2.1 Android version history

The first beta version of Android was released in November 2007 and about ten months later, the first Android smartphone, the HTC Dream (a. k. a. T-Mobile G1) was released together with Android 1.0, the first stable version of Android. In April 2009, Google started to release major versions of Android under an additional code name by assigning the name of a dessert to each version. Android 1.5, codenamed *Cupcake*, was the first version with such a name and all following code names are in ascending alphabetic order (e. g. *Donut* (1.6), *Eclair* (2.0/2.1), and *Froyo* (2.2)).

The last smartphone-only version of Android was 2.3 (*Gingerbread*) and the latest release of Gingerbread by the time we implemented our solution was version 2.3.7. In the first quarter of 2011, the first tablet optimized version of Android (codenamed *Honeycomb*) was released. Android's tablet version was released under the distinct version 3.x that allowed to distinguish the smartphone versions from the tablet versions.

The Honeycomb release divided the development tree into two separate branches: One branch was intended for smartphones and another one was optimized for use on tablet devices. This separation allowed Google to quickly release a version of Android that is optimized for the significantly larger screens of tablets; for example, it included a redesigned keyboard with additional keys [4]. Nevertheless, several tablets have also been released with one of the smartphone versions of Android, but this included some drawbacks in the interaction with the device due to the larger screen.

The latest major release of Android, version 4.0 (*Ice Cream Sandwich*), merged these two branches and it is the first version that officially supports both, smartphone and tablet devices. The source code of version 4.0.1 is publicly available on the Android Open Source Project (AOSP)[1]. Except for some proprietary drivers (e. g. Wi-Fi) and some Google-specific applications (e. g. the Android Market), the repository includes all files required to build the Android firmware. However, Google omitted to release the source code of Honeycomb at the time it was officially released and only made it available to device manufacturers. However, since the release of Ice Cream Sandwich, the source code of Honeycomb is now also part of the version history in the AOSP repository.

2.2 Android system architecture

The Android operating system builds on top of the Linux kernel and comprises several system libraries and a runtime environment, the Android Runtime. The Linux kernel is a free operating system kernel and it is the basis for all Unix-like operating systems. Due to its open source character, the Linux kernel is steadily improved by many contributors which makes it attractive as a basis for several operating systems.

The Android Runtime allows to execute applications in a predefined environment. As shown in Figure 2.1, Android provides an application framework that enables interaction with the operating system and allows an application to access the different functionalities of a device [7]. Android ships with several preinstalled applications (e. g. a web browser, an email client, a dialer application, and a contacts application), which provide the key functionalities of the mobile phone.

2.2.1 Linux kernel

The operating system itself builds upon version 2.6 of the Linux kernel. The kernel provides the required drivers to communicate with and abstract from the underlying hardware. It is, for example, responsible for network communication and for process and memory management. Further, the security mechanisms of the Linux kernel are used to enforce several of Android's security mechanisms.

2.2.2 System libraries

Several common C and C++ libraries are included in Android. They fulfill a large variety of different functionalities required by the Android middleware. These native

[1] Accessible at `http://source.android.com`.

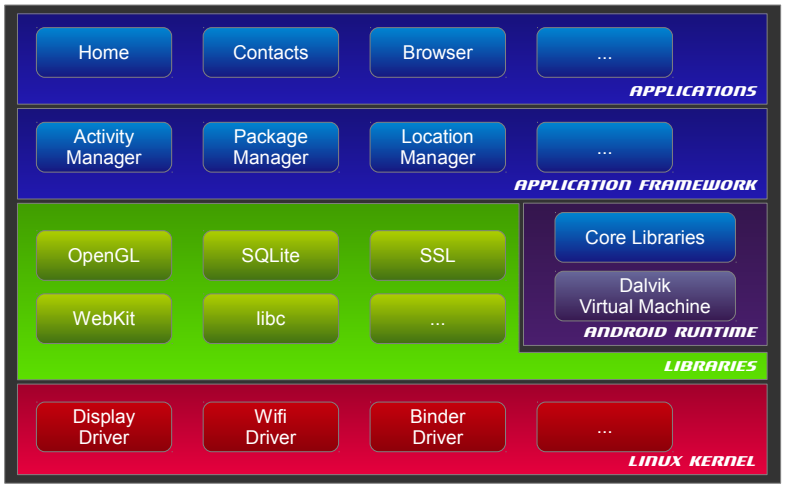


Figure 2.1: The system architecture of Android consists of five layers: A Linux kernel, several native libraries, the Android Runtime, an application framework and an application layer.

libraries mainly handle performance critical tasks since they can be performed faster using native code. This includes, for example, storing and retrieving data in a SQLite database, rendering web pages via the WebKit library, or establishing secure SSL network connections.

2.2.3 Android runtime

The Android Runtime consists of core Java libraries and a custom Java Virtual Machine called the Dalvik Virtual Machine (DVM). In contrast to native stack-based Java Virtual Machines, Dalvik uses a register-based architecture and is optimized for the limited capabilities of mobile devices [13]. These optimizations allow to run several instances of the DVM in parallel on a mobile device. The Dalvik compiler transforms the original Java bytecode (`.class` files) into a single `.dex` file that contains the Dalvik bytecode.

The core Java libraries of the DVM include a subset of the free Apache Harmony Java implementation and some custom libraries. The combination of these libraries neither fully supports the Java Standard Edition (Java SE) nor the Java Micro Edition (Java ME) platform [20]. Instead, the Java implementation for Android is highly adapted to the requirements of the Android operating system. As a consequence, applications developed for Java ME are not compatible with Android.

2.2.4 Application framework

Android's application framework is a middleware and runs on top of the Linux kernel. It is mainly developed in Java, but several parts of the framework use the Java Native Interface (JNI) to call native C or C++ code. In general, these native function calls

are used to access the underlying system libraries. The application framework itself is executed by the same Android runtime that is used to execute the applications.

The main purpose of the Android middleware is to provide all the functionality that is required to execute and to manage different Android applications. It is responsible for the installation of new applications, it undertakes all tasks that are required to manage the life cycle of an application, and it provides access to different features of the mobile device. To interact with the system, the Android middleware includes several services and interfaces. Applications can access those services and interfaces through an API. The API provides, for example, access to the GPS capability in order to retrieve the current geographic location.

2.2.5 Application layer

From the user's perspective, the key components of Android are the different applications that enable interaction with the device. Android ships with several integral applications that allow the user to access the crucial functionalities of a smartphone. This includes, amongst others, a dialer application to make a phone call, an SMS and an email application to send messages, a calendar application, and a browser to access the Internet.

Beyond these preinstalled applications, the user can further install third-party applications. These additional applications interact with the application framework through the same API and can in general access the same information and capabilities as the preinstalled applications. Since these preinstalled applications are not deeply integrated into the operating system, they can also be replaced by third-party applications.

2.3 Application Development

Android provides a software development kit, the Android SDK, which can be downloaded from the Android developers website[2]. It includes all tools and APIs to develop new Android applications. The major programming language for applications is Java. In addition, Android provides a Native Development Kit (NDK) which can be used to develop performance-critical parts of the application in C and C++. The native code can be called directly through the Java Native Interface, which permits direct interaction with the Java VM.

To ease the development of applications, the Android developers website also provides a plug-in for the Eclipse IDE[3]. The Android Development Tools (ADT) plug-in helps developers to initialize, implement, debug, and deploy new Android Applications.

As compared to the development of standalone Java applications, there exists one major difference in Android's application architecture. Android applications do not have a single entry point to start the application. Instead of calling a `main` method, the Android middleware can enter the application through four specific application components.

[2]Accessible at `http://developer.android.com`.
[3]The Eclipse integrated development environment (IDE) is available on `http://eclipse.org`.

These components accomplish distinct tasks:

- The *Activity* component for user interactions,
- the *Service* component performing background tasks,
- the *BroadcastReceiver* component listening to specific events (broadcasted by the system or by other applications), and
- the *ContentProvider* component for sharing an application's private data.

The Android middleware uses so called *intents* to handle the interaction between these components. These lightweight intent messages can be sent to activities, services, and broadcast receivers. The recipient of an intent is either *directly* mentioned by the recipient's fully qualified name or *indirectly* by an action string. In the latter case, the Android middleware selects a component that is capable of handling the desired action. The communication between different components can be limited by the developer and is accomplished by Android's permission system[4].

To inform the Android middleware about available application components and to provide further relevant information about an application, Android requires a configuration file, the `AndroidManifest.xml`.

2.3.1 AndroidManifest.xml

The `AndroidManifest.xml` file contains all information required by the Android middleware to install and run applications. It is an essential part of each application and includes, among others,

- the fully qualified - and unique - name of the application,
- the application's actual name and an optional description,
- a list of all application components the application consists of,
- all requested permissions the application requires to access protected functionalities, and
- a list of newly declared permissions to restrict the access to the application's own functionalities.

2.3.2 Inter-process communication / Binder

The Linux kernel supports several inter-process communication (IPC) mechanisms, such as pipes, shared memory and Unix domain sockets. In addition, Android uses a custom kernel driver, the *Binder*, to exchange data between different processes. The Binder kernel driver provides a lightweight service to exchange binary data between processes.

[4] Android's permission system is further described in Chapter 3.

Android's Binder implementation is based on the OpenBinder implementation by Hackborn [35]. All exchanged data has to be marshalled into a parcel on the sender side and is unmarshalled by the receiver. The required Java code for marshalling and unmarshalling data can be created automatically by using the Android Interface Definition Language (AIDL) [5]. The `aidl` tool shipped with the Android SDK allows to automatically transform an `.aidl` file into a Java stub class that implements the required Binder interface. However, the marshalling of data can also be done manually by writing the scalar values that should be transmitted into a parcel object.

2.3.3 Inter-component communication / Intents

Android's inter-component communication (ICC) mechanism uses *intent* messages to communicate between activities, services, and broadcast receivers. Intents basically serve two different purposes: Firstly, they describe the action to be performed, including all information required to accomplish it. These intents are used to start activities and services that can handle the desired action. Secondly, intents can inform broadcast receivers about specific occurring events.

The `Context` class of Android's application framework provides the `startActivity`, the `startService`, and the `sendBroadcast` method to trigger ICC. Each of these methods accepts an instance of the `Intent` class and the Android middleware forwards the given intent to the recipient component. The recipient of an intent can be set explicitly to a specific class. Explicit recipients are in general used to start known components, especially when they reside in the same application. This is the usual way to switch between the different activities of an application or to start an application's own service.

Additionally, the recipient of an intent can be addressed implicitly. Thus, the Android middleware tries to determine a suitable component that is able to handle the desired action. The implicit intent is identified by its *action string*. Action strings are simple names describing the action to be performed. An intent filtering mechanism in Android's middleware is used to decide which of the installed components are capable of handling the given intent. Therefore, the developer can specify several filtering rules for each component. These *intent filter* rules are defined in the `AndroidManifest.xml` file.

The implicit intent filtering provides a flexible way to manage several applications that can handle the same task. As an example, a user can install several web browsers on the mobile device. Independently of the installed browsers, an application can send an implicit intent to open a website. The Android middleware chooses the best application by using the system-wide default application for the specific action or by asking the user dynamically which application should be used.

The information about which website should be opened is part of the Intent message. Therefore, additional data can be written to and read from an intent in a key/value pair fashion. Further details about intents and especially about their filtering mechanism are concluded by Enck et al. [24] and by Burns [15].

2.3.4 Application components

Activity

Activities are the most important application components. Their main purpose is to interact directly with the user. Thus, they are the only components that have the ability to provide a user interface (UI). In general, each activity should only present one UI screen and should only fulfill one single task (e. g. listing all contacts). Related tasks that require another UI (e. g. to modify a contact) should reside in their own activity. Such a separation into different activities makes them reusable in different contexts.

An activity is implemented by extending the `Activity` class and overriding one or more of the methods that manage the lifecycle of the activity (e. g. `onCreate`, `onStart` and `onResume`). To start an activity it has to be registered in the `AndroidManifest.xml` file. Activities can either be started directly via the `startActivity` method of the `Context` class or indirectly by the Android middleware. Therefore, the middleware compares the action string of any given intent with the actions supported by the activity.

Service

Service components are meant to perform background tasks that do not require a user interface. However, they may start an activity component in order to interact with the user. Services are implemented by extending the `Service` class and overriding its `onCreate` method. A service can be started and stopped by other components via the `startService` and the `stopService` method of Android's `Context` class.

If further interaction with the service is required, the `bindService` method allows to receive a proxy object that allows to invoke methods of the service. The method also automatically starts the service if it is not already running. The communication to the service is established via ICC (in case the service and the caller reside in the same process) or via IPC (if both reside in different processes).

Broadcast Receiver

Android provides a mechanism to send and receive system-wide broadcast events. To receive such events, the *broadcast receiver* component can be implemented and registered for specific event types. The Android framework predefines several broadcast events (e. g. `android.provider.Telephony.SMS_RECEIVED` for incoming SMS), but developers may also define new broadcast events.

Again, a broadcast receiver is implemented by extending the `BroadcastReceiver` class. They register for specific broadcast events either via the `AndroidManifest.xml` file or programmatically by calling the `registerReceiver` method of the `Context` class. In both cases the intent filtering mechanism (see Section 2.3.3) is used to name the action string of the desired event.

Every application component can send broadcasts via the `sendBroadcast` method of the `Context` class by providing an instance of the `Intent` class. The intent contains the action

string that identifies the event and some additional optional information. Worth mentioning is the fact that broadcasts are transmitted to all registered receivers regardless of whether the receiving application is running or not. This allows to start applications on specific events, for example, after the device has finished booting.

Content Provider

The data of Android applications is, by default, not accessible to any other application. But Android allows to share the data of an application explicitly through a specific application component, the *content provider*. The access to the data is realized through a common interface. The interface defines basic CRUD[5] operations that have to be provided by each content provider.

Content providers are implemented by extending the `ContentProvider` class and they have to be declared in the `AndroidManifest.xml`. The manifest file defines the Uniform Resource Identifier (URI) of each content provider and all other applications can operate on a content provider through its specific URI. Access to the data can be restricted by protecting a content provider with additional permissions. In case a content provider is protected, other components have to request read or write permissions in order to be able to access the data. In addition, the content provider itself can grant fine-grained temporary access to the data based on the requested URI.

2.4 Application Deployment

Android applications are deployed within a compressed application package file (`.apk`). The archive contains the following files required to execute the application:

1. The compiled Java source code as Dalvik bytecode (`classes.dex`),
2. the compiled native source code if applicable,
3. the application's configuration file (`AndroidManifest.xml`),
4. all additional resources (e. g. images), and
5. a digital signature of the developer who signed the application package (`CERT.RSA` and `CERT.SF`).

This application package can then be installed on any device or uploaded to the Android Market for further distribution.

2.4.1 Android application stores

The Android Market is the official store to purchase and download applications for the Android platform. Since its opening in 2008 the number of applications has risen up to more than 200.000 applications in 2011 [10]. In the second quarter of 2011, Android

[5] CRUD stands for the four basic operations on persistent storage: Create, Read, Update, and Delete.

even overtook Apple's App Store with a market share of 44 %; compared to 31 % of the application downloads for Apple [1].

The Android market is directly linked to the user's Google account and can be accessed through a Market application that is preinstalled on many—but not all—Android phones. The integration of the Android Market is up to the device manufacturers and requires them to comply to Android's Compatibility Program[6].

In contrast to Apple's iOS, Android allows to obtain applications from alternative application stores. Thus, several additional sources for new applications (e. g. the Amazon Appstore, AppBrain, and SlideME[7]) exist beside the official Android Market.

[6] Accessible at `http://source.android.com/compatibility/index.html`.

[7] Accessible at `https://www.amazon.com/appstore`, `http://www.appbrain.com`, and `http://slideme.org`, respectively.

3
The Android security model

The extensibility of today's smartphone platforms gives the user the opportunity to extend the functionality of mobile phones through the installation of additional applications. To fulfill their task, these applications store more and more privacy sensitive data on the user's device. In particular, the user's e-mail conversations, private and business contacts, calendar entries and several login credentials are usually stored unencrypted on the device.

Vulnerabilities in the installed applications or even inside the underlying operating system put the private data at risk. To avoid such vulnerabilities at the forefront, Java is used as the primary programming language in Android. In contrast to other low-level languages like C, Java hinders developers from introducing buffer overflows and other vulnerabilities that can be exploited by an attacker to execute arbitrary code.

Further, Android uses several tools and techniques to defend against common control hijacking attacks in the Android operating system itself [2]. Control hijacking attacks often exploit vulnerabilities inside applications to place their malicious code on the stack or heap of an application. After attackers placed their code in the memory, they try to execute it by modifying the control flow of an application.

To make malicious code execution more difficult, Android protects the stack and the heap regions in the memory through the Non eXecutable (NX) hardware bit. This hardware-based protection denies executing code that is part of an application's stack or heap memory. As a consequence, attackers can not execute their malicious code in those memory regions. Additionally, the *ProPolice* extension for the *gcc* compiler is used to protect the stack against buffer overflows and the *safe-iop* library is used to detect integer overflows. And the mentioned techniques are only some examples of the implementation used to harden the security of the low-level operating system.

Android integrates several additional techniques in its framework that strengthen the security on the mobile phone. In the following, we first introduce the attacker model of

Android, and afterwards show the several techniques used to defend against the different security threats.

3.1 Attacker model

As we have seen, modern smartphones are an attractive target for attacks. However, to steal information or to infect the device with malware, attackers need access to the user's device. They may use one or even a combination of the following attacks to get access to the user's device and thereby to privacy sensitive data:

1. Attackers install malicious applications (e. g. through social engineering attacks),
2. they access private data by circumventing the existing security measures (e. g. by exploiting vulnerabilities on the device), and
3. they may get direct (physical) access to the device.

We categorize these three attacks into two categories: *Remote* and *local* attacks. The first two attacks, (1) and (2), are usually executed remotely over the network and therefore without physical access to the device. This approach allows to target a vast amount of devices without much effort for the attacker. The last attack (3) assumes that the attacker has physical access to the device. Getting direct access is not always feasible for an attacker and this approach does not scale to attack a large number of devices.

Malicious applications (1) are the most common security threat on mobile devices. In order to install the malicious application on the user's device, attackers often use social engineering techniques. They supply intentionally wrong information about the purpose of their application to trick the user into installing their malicious application. Afterwards, the malware may perform actions that are unintended by the user. For example, malware may read out the user's private data and transmit it to the attacker's server, or it may call costly telephone numbers.

Since modern smartphones are designed to be highly expandable through the installation of additional applications, they provide a variety of security measures to protect the phone. These protection mechanisms may restrict the access to several resources on the phone or they may reveal the purpose of the malicious application. Therefore, attacks try to exploit vulnerabilities in the operating system or in the installed applications to circumvent the security measures (2).

In rare cases, attackers may be interested in one particular phone and they may also have direct physical access to the device (3). If the phone is not protected by a screen lock, attackers have almost the same privileges as the original owner of the phone which allows to overcome several of the security mechanisms of the device. For example, they may install an application and grant it all required permissions or even modify the operating system itself.

Android's attacker model includes all three types of attacks and consequently defends against remote and local attacks. However, we assume that Android's estimate in the expertise of the average user are to high. Therefore, we will have a closer look at a refined

attacker model in Chapter 4, but we will first see how the current implementation of Android handles the existing security threats.

3.2 Protection techniques

The Android operating system makes use of several techniques to protect the device against the mentioned security threats. To mitigate against malware, Android uses application *sandboxing* and additionally provides a *permission system* that is based on mandatory access control. Together, these two techniques create a confined environment in which each application is executed. *Application signing* allows to compare the author of two applications such that malicious applications can not replace a trustworthy application on the device when the user installs a new or updates an existing application. Furthermore, the Android Market acts as a trustworthy[1] source for new applications.

The sandboxing approach and the permission model also reduce the impact of vulnerabilities inside an application, such that an exploited vulnerability may not lead to unlimited access to the device. The update mechanism of the Android Market also helps to deliver updates of an application more easily and more quickly. In addition, a simple update mechanism is used to deliver *firmware updates* to the end-user. This mechanism allows to deploy security updates of Android fast and easily.

To mitigate against direct access attacks, Android locks the phone with a *user-defined password* and recent versions of Android allow to encrypt the whole data partition of the mobile device.

3.2.1 Application sandboxing

The fundamental mechanism of Android's security model is a sandboxing approach to isolate running applications from each other and from the core system. The sandboxing approach is used to mitigate against remote attacks. Sandboxing has become important when dealing with untrusted content and is nowadays applied in several desktop applications, for example, in the Google Chrome web browser and in the latest version of Adobe's PDF Reader [54, 46].

In Android, process isolation is realized by assigning a unique[2] Unix user identifier (UID) to each newly installed application. This approach differs from the use of Linux' user model on a desktop machine. A desktop machine provides one user account for each physical user and almost all applications run with the privileges of the user that started the application. This implies that all applications run with the same privileges and are able to access each other's data. As a consequence, the user has to trust each executed application.

In contrast, Android's security model assumes that all applications are potentially malicious. Therefore, each application is started in a separate instance of the Dalvik VM. This instance runs with the permission of the assigned UID of the application.

[1] We will see later that this is—unfortunately—not always the case.

[2] Several applications may share the same user ID iff they are signed by the same certificate.

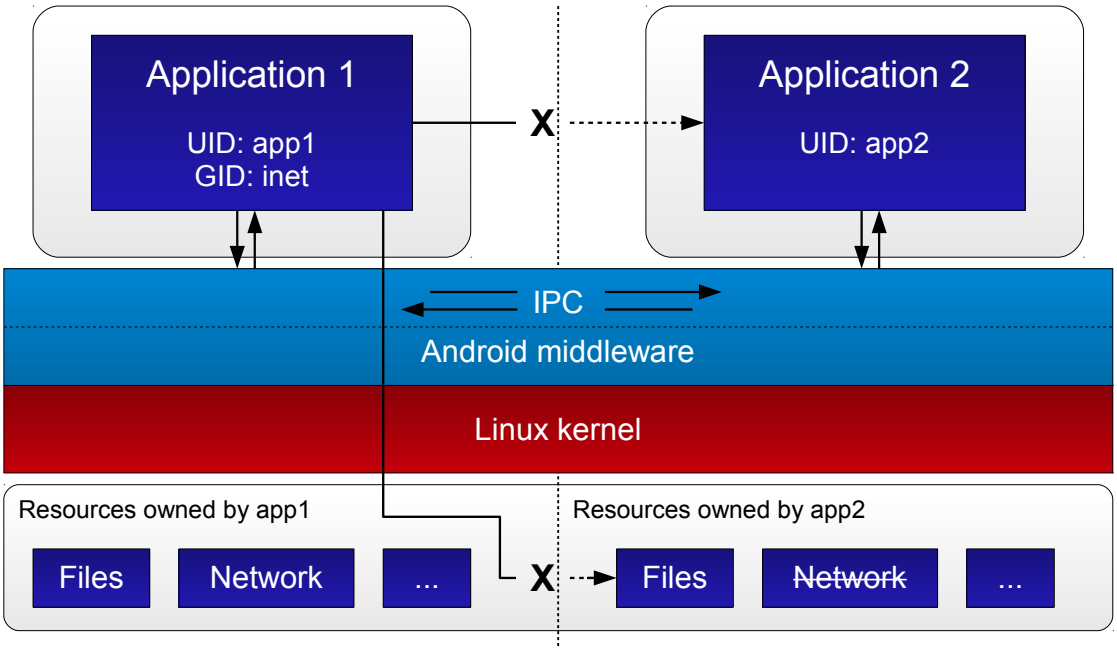


Figure 3.1: Each Android application is running in its own sandbox. Process isolation is enforced by the underlying Linux kernel which restricts the access to file system resources. IPC is possible through the Android middleware.

Hence, any data of an application that is stored in the file system, is owned by this specific *system* user and is, by default, not accessible to any other user. Additionally, all security critical system resources are owned by a distinct user. This approach protects the core resources of the operating system in the same way.

The discretionary access control is enforced by the underlying Linux kernel and helps to reduce the risk of a vulnerability inside an application. Figure 3.1 illustrates how the different resources of an application are protected. Hereby, a potential attacker should be prevented from gaining full control over a device by exploiting a vulnerability inside an application. However, Backes et al. [9] showed that this is not always fully true. They exploited a vulnerability inside the browser to install arbitrary applications. As a result, they compromised the integrity of the whole device.

3.2.2 Permission model

In addition to the sandboxing approach, the Android operating system uses a capability-based permission mechanism to restrict the operations an application is allowed to perform. An Android application is by default not allowed to access any of the device's low-level functionalities like determining the current location via GPS, establishing Internet connections, or letting the device vibrate. All these functionalities are available through a Java API, which is part of the Android SDK[3].

[3]The Android SDK and the API reference are available at `http://developer.android.com`.

Several of these API calls are protected by Android's fine-grained permission system. Only applications explicitly requesting a permission for a desired functionality are allowed to access it. Permission requests are defined in the `AndroidManifest.xml` file of an application by adding one or more `<uses-permission android:name="PERMISSION" />` tags with the desired permission name (e. g. `android.permission.INTERNET`) to the file. The Android operating system predefines several permissions[4], but any application can introduce new permissions to limit the access to its own functionality.

New permissions are also defined in the `AndroidManifest.xml`. They consist of a unique `name`, a `label`, a `description`, and a `protectionLevel` and are defined by the `<permission />` tag. The assigned protection level estimates the potential risk of a permission and decides whether the operating system grants a permission to an application or not. There are currently four different classifications:

- Permissions with a protection level of `normal` are always granted to the application and should not impact the security of the device.
- All permissions that enable access to private data or to security critical functionalities are classified as `dangerous`. For that reason, they have to be confirmed explicitly by the user at the time of the application installation.
- As with "normal" permissions, `signature` permissions require no approval by the user. However, they are only granted to applications that are signed with the same certificate as the application that declared the permission.
- `SignatureOrSystem` permissions are mainly meant to be used by system vendors. In addition to signature permissions, they are also granted to applications that have been signed with the certificate of the system image.

Figure 3.2 shows how the permissions are enforced through two different mechanisms. On the one hand, the Android middleware provides mechanisms to check permissions implicitly and explicitly; on the other hand, the Linux kernel enforces several permissions that protect low-level functionalities.

Permission enforcement by the Android middleware

In conjunction with the definition of application components in the `AndroidManifest.xml`, each application component can be protected with one optional permission. The Android middleware ensures implicitly that only callers with this particular permission are allowed to interact with the protected components. Consequently, the permission of the caller is checked whenever it starts an activity; whenever it starts, stops, or binds to a service; or whenever it reads from or writes to a content provider.

Likewise, senders of a broadcast can only notify a protected broadcast receiver if they possess the appropriate permission, otherwise the broadcast is not delivered. But when

[4] See `http://developer.android.com/reference/android/Manifest.permission.html` for a list of predefined permissions.

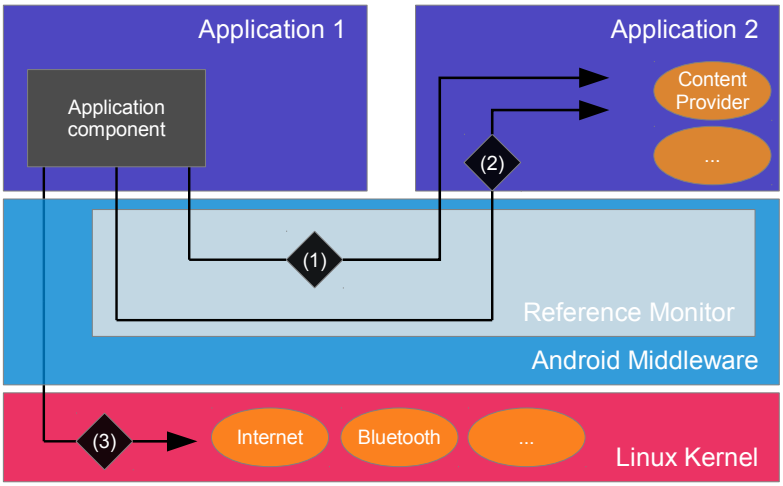


Figure 3.2: Permissions are enforced implicitly by the Android middleware (1), explicitly inside the application (2), or by the Linux kernel (3).

sending broadcasts a converse restriction can also be set. The sender can programmatically define a permission that is required to receive the broadcast. In this case, only receivers with this particular permission will get notified.

The Android middleware also allows to check the permission explicitly. Therefore, different methods (e. g. `checkCallingPermission()` or `checkPermission()`) of the `Context` class can be used to check whether the calling process has been granted a certain permission.

Permission enforcement by the Linux kernel

Several of the core permissions in Android rely on the security concepts of the Linux kernel. These core permissions, which mainly protect low-level functionalities, are enforced by making use of the discretionary access control (DAC) mechanism of Unix.

Each resource on the Unix file system has an assigned file mode that describes the operations allowed on the resource. The potential operations are *read*, *write*, and *execute* and they can be set separately for three different user classes: *user*, *group*, and *others*. Each file resource is additionally owned by a specific user and by a specific group. Consequently, the file mode implies which operations the owner of the file is allowed to perform, which operations members of a specific group are allowed to perform, and which operations are allowed to all other users.

In Android, certain low-level devices and libraries on the file system are owned by distinct Unix groups. These groups are allowed to perform the required operations on the respective resource (e. g. to read and write to a device or to execute code inside a library); all other users are not allowed to access the resource. These Android specific Unix groups are directly linked to specific Android permissions that protect access to low-level functionalities. If an application requests one of the permissions and the user acknowledged it, the Android middleware adds the Unix user of the application to the

corresponding group. As a consequence, the Linux kernel grants the right to access the protected resource.

3.2.3 Application signing

To install an application on a device, the application package file (`.apk`) has to be signed by the developer. The purpose of this signing process is to identify applications that have been created by the same developer. In contrast to other platforms (e. g. Symbian), the certificate used to sign an application can be self-signed and must not be signed by a trusted authority. Nevertheless, this signing approach has three positive consequences for the security of the Android operating system.

First of all, the certificate of an application is checked whenever an installed application gets updated to a newer version. The update only proceeds if the certificate of the new version matches the one of the installed version; if they do not match, there is no way to replace the old version of the application automatically. The user will either have to manually remove the old version of the application before installing the new one or the new version must be released under a different package name. In the latter case, the new version will be installed in parallel to the old version.

Secondly, the signing of applications with the same certificate gives the developer the possibility to run two applications in the same sandbox. For this purpose, a developer can request the same UID for several applications by defining the same value for the `sharedUserId` attribute in the `AndroidManifest.xml` files. Further, the `process` attribute enables the developer to put several applications into the same process. Applications sharing the same UID are treated as one unit and consequently, they share the union of all requested permissions. To avoid that malicious applications bypass the security checks of the Linux kernel by declaring the same UID as an already installed application, these UID requests are only accepted for applications that have been signed with the same certificate.

Thirdly, the signing process allows a developer to restrict the inter-process communication with respect to Android's permission model. By setting the protection level of a new permission to `signature`, the Android middleware only grants this permission to applications that have been signed by the same developer. As a consequence, two or more applications from the same developer can communicate via Android's inter-process communication mechanism without revealing data to any other application.

The signing mechanisms again defend against remote attacks in which the attacker tries to install a malicious application on the user's device. The attacker can not remotely replace an installed application with a malicious one.

3.2.4 Centrally managed application deployment

As other smartphone systems, Android provides a centrally managed application store. The Android Market is the official source to download and install new applications. In contrast to desktop computers, where software is often downloaded from different untrusted sources, the centralized repository reduces the risk of installing malicious ap-

plications. Further, the Android Market includes an update mechanism to keep all the installed applications on a device up-to-date. This mechanism allows to close security holes inside applications more quickly. Consequently, the Android Market helps to mitigate malicious applications and thus protects against remote attacks.

In contrast to other mobile platforms (e. g. Apple's App Store), the Android Market does not apply any approval process for uploaded applications [8, 33]. Any registered developer can upload applications to the Market and these applications are available almost immediately without being screened for malicious behavior. To identify poor or even malicious applications, the Android Market uses a user review system. This system allows users to rate each application with one to five stars. Google assumes that the review mechanism will identify poor applications quite quickly and therefore restrain further users from installing the application. It is a debatable point whether this approach can identify malicious applications, because the general user is no security expert. Additionally, this approach accepts that the first couple of users install the application without any auxiliary information about its quality.

Another protection mechanism against spam and low quality applications in the Android Market is a registration fee of USD 25 to create a new Android developer account. The registration fee makes it unprofitable for an attacker to scatter applications under many different pseudonyms. Thus, as soon as one infected application has been identified, Google can easily find additional infected applications by examining all applications that correspond to the same developer account.

Due to the missing approval process, several malicious applications have been found in and were removed from the Android Market [53, 16]. Android malware relies on one or several dangerous permissions to fulfill its task. It therefore promises attractive features and tries to trick the user into granting the application a combination of dangerous permissions. As a consequence, malicious applications are able to steal the user's private data, collect location information, subscribe to costly premium services, or make the phone part of a botnet.

In March 2011, several applications in the Market were infected by a Trojan named *DroidDream*. The malware was included in over 50 applications and had been downloaded between 50.000 and 200.000 times before it was finally removed from the Market. Instead of tricking the user into granting an application dangerous permissions, DroidDream exploited a vulnerability inside Android. Using this vulnerability, DroidDream rooted the user's phone and silently installed a backdoor. By this time, the vulnerability was actually already fixed in the latest Gingerbread release of Android. Unfortunately, not everybody was able to update the mobile device to that latest Android version.

It is unlikely that the malware situation for Android will become better. For the second quarter of 2011, McAfee Labs [40] even reported that over 50 % of new mobile malware threats targeted the Android platform. This is not really surprising since the market share of Android has been growing rapidly.

3.2.5 Android version updates

Exploitable vulnerabilities inside the Android system itself harm the security of the user's device dramatically. Therefore, it is important that security fixes are rolled out as soon as possible. Google releases security updates quite quickly and also integrates the security fixes into the Android Open Source Project, such that firmware modifications can also close the security holes. To ease the installation of the firmware for the end-user, updates are installed Over the Air (OTA), which means that a network connection is sufficient to retrieve and install the new firmware. As a consequence, even users without profound technical skills will be able to keep their devices up-to-date. The update mechanism helps to reduce the risk that an attacker can exploit vulnerabilities and therefore mitigates remote attacks on the device.

Nevertheless, there is one downside when we compare Android phones to Apple's iPhone: Apple's closed infrastructure makes it easy to keep each iPhone model up to date, because the update is directly shipped from Apple to the customer's phone. However, this is not true for Android. Depending on the phone, Android updates have to pass through phone manufacturers and maybe also phone carriers before they eventually arrive on the customer's device [19]. As a consequence, over 50 % of the devices have not been updated to the latest Honeycomb release and are running an Android version that is one or even more major versions behind [6].

To improve this situation, Google formed, together with several phone manufacturers and carriers, the *Android Update Alliance* [55]. Regrettably, few details were announced about their agreement; the only detail published is the intention to provide updates for a period of at least 18 months. A renewed view on the situation three months after the announcement showed that several phones received an update [50]. Nevertheless, the situation still needs further improvements.

3.2.6 Password protection

Android allows to protect the device through a user-defined password. The user has to supply this password whenever the phone is started and whenever the device screen is locked. If the user sets a timeout to automatically turn off the screen, the device will automatically be locked and will ask for the password to unlock. The password can be supplied as a pattern of nine connected dots[5], as a numeric PIN, or as a full alpha-numeric password.

This mechanism prevents direct interaction with the phone if the phone gets lost or in case it is left unattended. Thus, this password requests defend against local attacks against the device. However, an attacker can still remove the SD card of the phone and read out its data with an external card reader.

[5]The user has to connect nine dots in the right order by sliding over them.

3.2.7 File system encryption

Since Android 3.0, Android supports rudimentary encryption of the device's file system. We call the current state of Android's encryption mechanism rudimentary, because only the `/data` partition of the SD card is encrypted; all other files (e. g. pictures) are still stored unencrypted. However, the encryption of the data partition secures the information that is stored in the application specific directories. This may, for example, include database and text files with crucial information (e. g. login credentials).

The data is encrypted using the Advanced Encryption Standard (AES) block cipher with a key size of 128 bits; the cipher operates in cipher block chaining (CBC) mode and the initialization vector is generated via Encrypted Salt-Sector Initialization Vector (ESSIV) with the SHA256 hash function [3, 27]. The user-provided password is used to encrypt the secret encryption key, which is used for the file encryption. In combination with the password protection mechanism, the file system encryption partially protects against direct access by an attacker even if the attacker is able to get direct access to the SD card. Therefore, the encryption of the data partition helps to protect against local attacks mounted against a device.

4

User-controlled Internet connections in Android (Ucicia)

Although the security model of Android includes several different security mechanisms, we see more and more malware aiming at the Android platform. Many malicious applications utilize the network functionality of the device which is explicitly protected by Android's permission system. Nevertheless, this protection mechanism does not seem to be very effective to mitigate malware that relies on network access. The problem is that the permission to access the Internet is requested by too many applications and that the average user can not estimate the risk that arises from granting an application the right to access the Internet.

The Internet permission can be abused in many ways; for example, by making the device be part of a botnet or by stealing the owner's personal information. The latter especially becomes very security critical if the stolen information can be used to steal money from the user. A simple example would be a malicious banking application that tries to steal user credentials in order to later conduct unauthorized transactions.

In January 2010, at least 39 unofficial banking applications were uploaded to the Android market by the same user [28]. They targeted customers of several international banks. It seems as if the only purpose of these applications was to trick some people into buying the $0.99 application, whereas the only functionality of the applications was to open the official online banking website. Nevertheless, the application's request of the Internet permission would have been sufficient to steal the user's credentials and to transmit them without user notice to the server of the attacker.

However, requesting Internet access is quite natural for an online banking application, and unfortunately, the Internet permission request on Android does not reveal to which destination the application wants to connect to. To mitigate against malware threats that heavily rely on the Internet permission, we extend the security model of Android

and give users more control over the Internet connections on their device. We let the user decide whether the Internet permission should be granted or not and provide the opportunity to limit the Internet access to a restricted subset of destinations.

Before we describe our contribution to the Android operating system, we first introduce the underlying network basics that are used in this thesis and provide an introduction to different access control mechanisms. Afterwards, we show the current weaknesses of Android with respect to the Internet permission. Based on these weaknesses, we refine the attacker model against Android. We close the chapter with a high-level overview of our solution and show how it is possible to defend against the new attacker model.

4.1 Basics

Our solution refines the Internet permission of the Android operating system and enforces this new permission model on the device. In order to refine the existing permission model, it is necessary to understand the underlying network basics that are used when a connection with the Internet is established. Further, the enforcement of the modified permission system requires an understanding of available access control policies.

4.1.1 Network communication

Since this work targets the Internet permission, our proposed solution is dependent on the underlying network transport mechanisms. The interconnected machines of such a network communicate with each other through several protocols. We will discuss the most important protocols that are required in this thesis. Further, we will have a look at the Domain Name System that allows to identify a host in the network by its hostname.

Internet protocol stack

In order to establish a communication between two machines that are interconnected by a network of several machines, the following tasks have to be accomplished: First of all, the two communicating machines have to agree on a unique message structure, such that both are able to understand the message. Secondly, the arbitrarily long message is split into chunks of the same size and has to be transmitted from one machine to the other machine in a way that the receiver is able to rebuild the original message—even if chunks arrive in the wrong order. Thirdly, different machines need to be uniquely identifiable in order to transmit the message to the correct destination machine. Fourthly, the machines are connected through a wire or through another communication medium and need to transmit the message to the desired destination; and fifthly, the machines in the network need to exchange information by transmitting the individual bits of the message on the hardware basis.

The Internet communication is realized by using several protocols, which each undertake one of the above mentioned tasks. These protocols are categorized into five distinct layers. Together, they form the *Internet protocol stack* [38]. As applicable in Figure 4.1, the five layers are

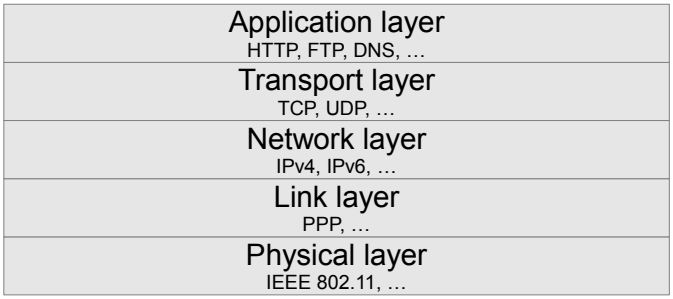


Figure 4.1: The Internet protocol stack organized in five layers.

1. the application layer,
2. the transport layer,
3. the network layer,
4. the link layer, and
5. the physical layer.

Each of the layers fulfills a specific task in the communication. The categorization into different layers allows to easily exchange the used protocol of one layer without influencing the task of any other layer. For example, the same transport and network protocol can be used to transmit different types of messages (i. e. different application protocols are used); but it is also possible to send the same message over a wired or a wireless network infrastructure without changing the transport or network layer protocols.

The link and the physical layer are very hardware oriented. Therefore, we omit further details about these two layers, since the actual transfer of the bits is not relevant for this thesis. In the following, we will focus on the first three layers and especially on the most important protocols of these layers. We start with the IP protocol of the network layer and proceed bottom-up.

Internet Protocol The Internet Protocol (IP) is used to transport network packets between two different machines. Different machines are addressed by a unique identification number, called the *IP address*. In the Internet Protocol version 4 (IPv4) the IP address is a 32-bit integer value that is commonly represented in dot-decimal notation (e. g. 127.0.0.1). IP packets contain both the source IP address of the sender and the destination IP address of the receiver. This information allows to route packets to their desired location.

Transport Protocols (TCP and UDP) The Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP) are transport layer protocols used to transmit a stream of bytes from one machine to another. It enhances the Internet Protocol in such a way that different packets can be assigned to specific connections. TCP establishes

a reliable and ordered connection between two machines and guarantees that the byte stream arrives in the same order and without any gaps, whereas UDP is a simpler protocol with less overhead. Consequently parts of the byte stream may arrive in a different order, may appear twice, or may be lost without notice.

Both protocols include two 16-bit *port numbers* to identify the endpoints of the communicating machines. Depending on the destination port, the operating system forwards the incoming packet to different applications.

Application Protocols (HTTP) One of the most important application protocols on the Internet is the Hypertext Transfer Protocol (HTTP). It is used for the communication between a web browser and a web server. HTTP connections are established via TCP and the port used for communication on the server side is port 80.

As we have seen before, several domain names can point to the same IP address and consequently to the same web server. Since none of the underlying protocols (neither IP nor TCP) includes the domain name, the header of an HTTP request includes the *Host* field with the requested domain name (e. g. www.google.com).

Domain Name System

The use of 32-bit integer numbers to set the destination of a packet is, from a user's perspective, rather cumbersome. Therefore, a hierarchical Domain Name System (DNS) is used to address a machine in the network via a meaningful domain name (e. g. www.google.com). The hierarchical network of DNS servers is capable of translating a given domain name back into an IP address.

Important to notice are three facts: Firstly, the result of a DNS request is not a single IP address, but may be a set of those. Secondly, several different domain names can point to the same IP address. And last, several DNS requests may not return the same set of IP addresses. From this it follows that the mapping between IP addresses and domain names is an $N : M$ relation.

4.1.2 Access control mechanism

There exist several access control mechanisms to restrict the access to a resource on a computer system. In the following part, we will have a closer look at the three most important access control mechanisms. Thereby, we concentrate on the possibility to restrict the Internet access of the several applications that run on Android in a fine grained manner.

In general, the access control policy decides whether a *subject* is allowed to perform an *operation* on an *object*. Therefore, we define the *subject* of the access control policy to be the instance that wants to access a resource (e. g. a user); we define the *object* as the resource being accessed (e. g. a file); and we define the *operation* as the operation the subject wants to perform on the object (e. g. reading or writing).

Mandatory access control

For mandatory access control (MAC) the decision whether an operation on an object is permitted or not is based on properties of the object and the subject, and on several global rules that act upon these properties. Consequently, the actual identity of the subject is irrelevant for the decision. MAC is often mentioned together with military multi-level secure systems. These military systems handle documents (objects) with multiple security classifications in one system [49]. Every user (subject) and every document has an assigned security level of: Unclassified, Confidential, Secret, or Top Secret. The MAC mechanism now ensures that no user can read documents with a higher security level than the own level.

This restriction is achieved through two rules: The first rule restricts the reading (first operation) of documents and states that users are only allowed to read documents with the same or a lower security level. The second rule restricts the writing (second operation) of documents and only allows a user to write documents with the same or an higher security level. The purpose of the first rule is quite obvious; the second rule however ensures that a document can not be made available to a lower security level.

This example shows that it does not matter at all which subject wants to access an object; the access control only relies on the global rules. This access control policy is named *mandatory*, because although an entity may have access to a specific resource it can not pass this access privilege to any other entity [47].

Discretionary access control

In contrast to MAC, discretionary access control (DAC) is based on the identity of the subject that wants to access an object. DAC is the mechanism that is used in the Unix and the Windows operating system to restrict the access to the different resources of the computer. For each resource exists an access control list that defines which operations (e. g. read, write, and execute on Unix) are allowed by which user.

Consequently, one could represent DAC as a three-dimensional matrix of user, resource, and operation: Each cell contains a boolean value that determines whether the operation by the user on the object is permitted or not. This access control policy is named *discretionary*, because an entity can have permission to grant other entities access to a specific resource [47].

Role-based access control

Another access control mechanism is role-based access control (RBAC). RBAC allows to define *roles* that can be assigned to several subjects. The role-based approach helps to reduce the administration overhead by assigning the desired properties to all subjects with a specific role, instead of defining it for each subject separately. RBAC can be used both with MAC and with DAC.

Both Unix and Windows use RBAC together with DAC. They allow to define so called groups that can contain several users. These groups can then be used to allow all members specific operations on a system resource.

4.2 Refined attacker model

As seen in Chapter 3, Android provides a security model that is based on mandatory access control. The permission system of Android is used to decide whether an application can access a specific functionality or not. One of the permissions that is used by many applications is the Internet permission (`android.permission.INTERNET`), which is required to establish network connections.

The Internet permission is enforced by adding the Unix UID of an application to a specific Unix group, named *inet*. Without this group membership, applications are not allowed to open new network sockets. Consequently, the granularity of the Internet permission is rather coarse and the permission only support two states: *Full* Internet access or *no* Internet access at all.

Network connections can be misused in several ways and imply a high security risk. Therefore, they are classified as *dangerous* permissions and the user has to explicitly grant applications the permission to access the Internet. Once the user has confirmed the permission request, which is done during the application installation, the application can access the network without further approval. Revocation of any permission is not possible, or rather implies the uninstallation of the application.

As shown by Barrera et al. [11], the Internet permission is the most frequently requested permission. Over 60 % of their analyzed applications requested to access the network. They argue that this is due to the Internet-orientated services of these modern smartphones and due to the fact that many free applications make use of mobile advertisement. Consequently, it is quite normal that an application requests the Internet permission and many users don't consider whether the Internet permission request is really necessary.

Attackers can utilize this fact and request the dangerous Internet permission without causing suspicion. Afterwards, they can abuse the Internet permission in several malicious ways, for example,

1. to receive and execute malicious code,
2. to remotely control the device, or
3. to steal the user's private data.

For (1) and (2) the attacker only has to convince the user that the Internet permission is required for an application. As we have seen, it is quite hard for users to estimate the risk that is associated with the Internet permission and to decide whether a network connection is really mandatory to execute the application. For (3) the attacker has to additionally convince the user that the request to access the user's private data is plausible.

Information gathering The private data that is stored on the user's device is quite diverse. There is device related information like the IMEI and IMSI number that allow to uniquely identify the device and the SIM card. Additionally, the device contains the

user's private information like contacts and message history, but also security critical login credentials are stored on the device.

If an attacker wants to steal private data, there are two possibilities to obtain the information:

1. The data is available through a public API by another application (e. g. contacts) or

2. the malicious application requests the data itself (e. g. login credentials).

In both cases, the malicious behavior is hard to distinguish from normal behavior. In the first case, the API may or may not be protected by Android's permission system such that the API access may entail a permission request by the attacker.

However, if the information is obtained through an unprotected API, there is no way to recognize it, but even if the API is protected by a *normal* permission, it is hard to notice. For this reason, the private information that is defined in Android's core functionality (e. g. contacts) are protected by a *dangerous* permission. However, attackers can try to add some related functionality that makes the permission request plausible. Consequently, accessing publicly available private data through an API is feasible for an attacker.

For the second case, when the application requests the information itself, Android's permission system can not protect the user from entering private information. However, the attacker has to convince the user that entering the information is mandatory and safe. One way to do so, is to mimic some official application. This happened, as an example, to the *Netflix* application that allows to watch movies on the smartphone. The Netflix application was available in the official Android Market; however, only to a limited number of devices. This made it possible to distribute a malicious version of the Netflix application outside the official Market. The application looked similar to the official application, but stole the login credentials of the user and transmitted them to a remote server [39, 52].

Attacker model Android's current attacker model does not mitigate against attacks that try to use the Internet connection in an unintended manner. Therefore, we extend the attacker model as follows: Attackers still have no physical access to the device and we assume that Android's protection mechanisms (sandboxing and the permission system) can not be circumvented. Consequently, attackers can not get additional permissions by exploiting potential existing vulnerabilities. As before, attackers can upload malicious applications to the official Android Market.

However, we assume that the average user is able to estimate the risk of granting *full* Internet access, so that they will not install any application that requests full Internet permission without a comprehensible reason. In addition, we assume that a user can judge on the requested connections and decide which are reasonable and which are suspicious.

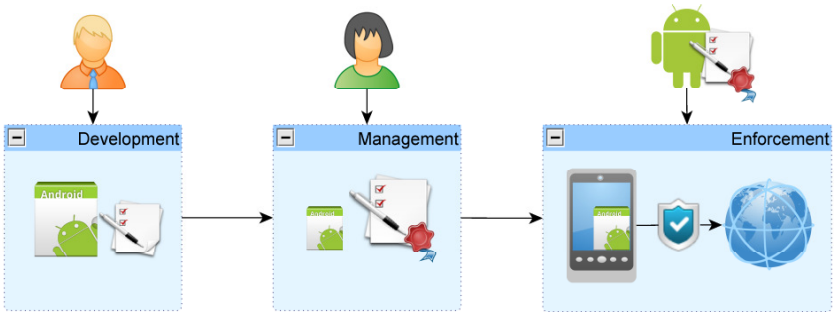


Figure 4.2: High-level overview over the Ucicia modifications.

4.3 Ucicia: A high-level overview

We provide a solution to this refined attacker model, named *Ucicia* (user-controlled Internet connections in Android). Therefore, we modified the Android operating system and enhanced the security mechanisms of Android.

As applicable from the high-level overview in Figure 4.2, our modifications affect three parties that interact with the application: Firstly, we give application developers the possibility to request fine-grained Internet permissions. Secondly, users grant or refuse the requested Internet permissions at the time of the application installation and they are also able to manage them later; and thirdly, the Android operating system enforces the new fine-grained Internet permissions.

In order to enforce the new Internet permission, we had to choose an appropriate access control policy. The task was defined as follows: We want to control whether an application (*subject*) is allowed to access a specific Internet location (*object*) or not.[1] We extended Android's mandatory access control mechanisms and give the user the control to define the global rules that all applications have to act on. Applications themselves shall not have the possibility to pass their permission on to any other application.

Since Android directly assigns each installed application a distinct Unix UID, we can define a set of restriction rules per UID—and consequently per application. To restrict the Internet access for each application separately, we stick to Android's enforcement mechanisms and make use of the discretionary access control mechanisms of the Linux kernel. However, it is important to mention that none of the third-party applications can either change existing Internet permission rules or pass the granted Internet permission on to another application. Our final solution is separated into two parts:

1. A *refined permission model* that allows fine-grained Internet permissions for outgoing connections and

2. our *Ucicia implementation* that manages and enforces this new permission model.

We will discuss the modified permission model of Android in the next chapter and show the implementation of Ucicia in Chapter 6.

[1] Hence, we only control one *operation* that judges upon outgoing Internet connections.

5
Permission model refinement

As shown in the last sections, the security of Android is weakened by its coarse definition of the Internet permission. This is especially true in case this permission is granted in conjunction with further permissions that give the application access to the user's private data. The consequences that arise from such a combination of permissions are difficult to be grasped by the average user.

To make the consequences more obvious and to give the user more control over the Internet connections of the device, we extend Android's permission system with respect to the Internet permission. Instead of granting full Internet access, we allow to restrict the network communication to a limited subset. We give both the developer and the user of an application additional tools to control the network communication. Our refinement of the Internet permission consists of five key functionalities:

1. *Fine-grained Internet permission requests* by limiting network connections to specific destinations.

2. *Refinement of requested Internet permission* that allows to modify requested Internet permissions at the time of application installation.

3. *Granting and revoking permissions at any time.*

4. *Dynamic Internet permission* requests that allow an application to request the Internet permission at runtime.

5. *Observation of Internet connections* to identify malicious behavior.

In the following chapter, we will define a mathematical model for our extension of Android's security model. Afterwards, we present the definition of each of the five key functionalities in detail.

5.1 Definitions

We provide several definitions that describe an Internet connection in a mathematical manner. These definitions are based on the underlying network architecture that we discussed in Chapter 4. Using these definitions, we define the set of all possible Internet connections an application can establish. Finally, these definitions allow us to define the restricted Internet access of the *user-controlled Internet connections*.

5.1.1 IP addresses, domain names, and ports

We define the set of all possible IPv4 addresses I, the set of all possible domain names D (with respect to [43, section 3.5]), and the set of valid ports P as follows:

$$\begin{aligned} I &:= \{0, \ldots, 2^{32} - 1\} \\ D &:= \{d \mid d \text{ is a valid domain name}\} \\ P &:= \{0, \ldots, 2^{16} - 1\} \end{aligned}$$

5.1.2 Outgoing connections

Outgoing TCP and UDP connections are identified by their destination IP address and the destination port. Based on the definitions above, we can define the set of all possible *outgoing connections* as a set of tuples:

$$C := \{(i, p) \mid i \in I \wedge p \in P\}$$

Further, we define the set of permitted outgoing Internet connections for an application *app* to be

$$C_{app} \subseteq C$$

As mentioned earlier, Android only supports two different states for the Internet permission. Either an application has *full Internet access* (i. e. $C_{app} = C$) or *no Internet access* at all (i. e. $C_{app} = \emptyset$). Our extensions to Android allow to define an arbitrary subset of permitted connections for each application. Consequently, an application may only have *restricted Internet access*.

5.1.3 Restriction rules

To limit the Internet access of a device, we define the set of all possible *restriction rules* as a tuple of hostname (which is either an IP address or a domain name) and port:

$$R := \{(h, p) \mid h \in I \cup D \wedge p \in P\}$$

This allows us to define IP-based and domain-based restrictions combined with a destination port for each application *app*:

$$R_{app} \subseteq R$$

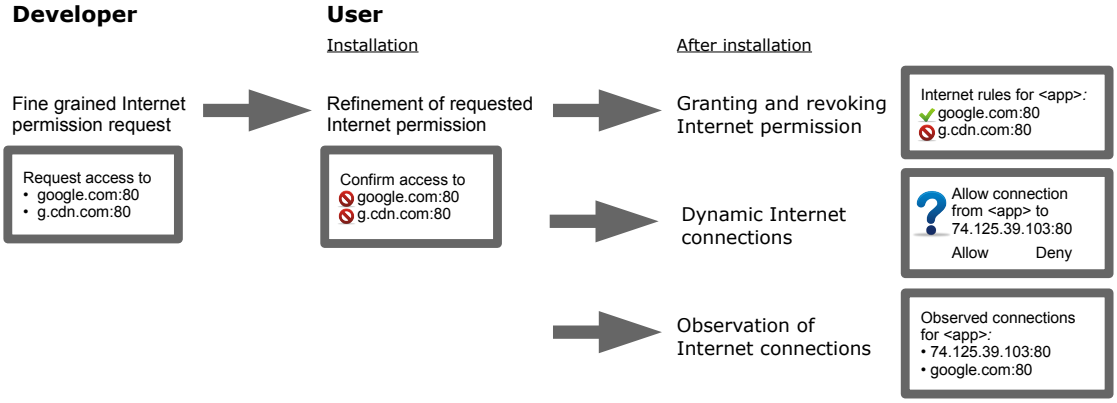


Figure 5.1: Our approach includes five key functionalities that affect both developers and users of an application.

The DNS mechanism can be simulated via an abstract lookup function $l_{DNS} : D \to 2^I$ that maps a set if IP addresses to each domain name. Using this function, we are able to get the set of allowed connections from a set of restriction rules via a function $u : R \to C$ that is defined as follows:

$$\begin{aligned} u(R_{app}) \quad := \quad & \{(i,p) \mid (i,p) \in R_{app} \wedge i \in I \wedge p \in P\} \cup \\ & \{(i,p) \mid (d,p) \in R_{app} \wedge d \in D \wedge i \in l_{DNS}(d) \wedge p \in P\} \end{aligned}$$

5.2 New key functionalities

The provided definitions restrict the network connections that are allowed by an application. However, what is still missing is a way to define, modify, and enforce these restrictions on the mobile device. In the following, we will present the five key functionalities of our solution. These functionalities allow the developer and the user to restrict the outgoing Internet connections per application.

Our modifications affect three stages of an applications lifecycle. Firstly, we modify the development process and give the developer the possibility to request fine-grained Internet permissions. Secondly, the user can modify the requested Internet permission when the application is being installed; and thirdly, the user can modify the Internet permission of any application at any time after the application has been installed. Figure 5.1 provides a short example for each of the five refinements and shows where the different changes apply.

5.2.1 Fine-grained Internet permission requests

Android's permission design expects that the *developer* of an application requests all permissions required to execute the application. Our solution does not change this approach and leaves the responsibility to the developers, with the result that they must determine the hosts their application communicates with.

Thus, our solution allows developers to define a set of restriction rules, as defined in Section 5.1. The rules are defined as a list of IP addresses or domain names together with an optional destination port. If no port is provided, we allow connections to all possible ports. For most developers the number of hosts is relatively small such that the effort to define each host separately is negligible. However, developers may increase the confidence of users into their application by giving the users an insight into the set of allowed network connections of the application.

The provided restriction rules are automatically enforced by the operating system. In case the developer decides to request *no Internet access* or *full Internet access*, nothing changes for the developer. Consequently, our solution is totally backwards compatible for the developer.

5.2.2 Refinement of requested Internet permission

Similar to the extensions we provide to the developer, we want to give the user more control. First of all, whenever a user installs a new application, it should be possible to review the hosts the application wants to communicate with. This gives the user more background information on whether the application is trustworthy or not. As a by-product, the list of hosts may also reveal whether an application uses advertisements.

Android's permission system is currently not capable of installing an application without granting it all requested permissions. If the user is not satisfied with the requested permissions of an application, the only possibility to deny permissions is to cancel the installation process and to abandon the application. We relax this policy with respect to the Internet permission and allow users to install applications that request the Internet permission without allowing the applications to access the Internet. In comparison to other permissions, revoking the Internet permission should not crash a well programmed application for several reasons.

Firstly, the Internet permission is not checked explicitly by the Android middleware and does not raise any security exceptions in case the permission is not present. Secondly, our solution grants the Internet permission itself in the first place, but denies future access to the network by filtering out all network packets of the application. Such a situation is similar to the situation in which the network functionality of the device is temporarily disabled. No network access is a situation that should be handled by developers anyway.

5.2.3 Granting and revoking permissions at any time

In addition to the revocation of the Internet permission at the time of installation, we allow the user to review and manage the Internet access of an application at any time. Therefore, we provide a flexible tool to manage the permitted Internet connections for each application. The tool lists all the hosts predefined by the developer and allows to change the state for each of these destinations separately.

Similar to current personal desktop firewalls, the user can decide whether a specific connection should be *allowed* or *denied* . Further, it is possible to postpone the decision

to the time when the connection is actually established.[1] Consequently, we allow three states for each Internet connection: *ALLOW*, *DENY*, and *ASK*. The user can also define the default behavior for all connections that are not handled by one of the already defined rules. This approach allows both, blacklisting and whitelisting of specific Internet connections and thereby gives the user full control over the possible connections of an application.

Further, we give the user the possibility to create new rules that restrict or extend the Internet permission of the application. The new rules are again based on the IP address or the domain name with an optional destination port.

5.2.4 Dynamic Internet Permission requests

Android's permission system is rather static and demands the decision whether a permission is granted to an application or not before the permission is actually needed. We enhanced Android's permission system and allow a *dynamic Internet permission* request that asks the user for the permission when the Internet connection is actually established. The user can enable this dynamic Internet request for each connection of an application separately.

The desired Internet connection is intercepted by the operating system and the user receives a notification about the queued connection. The user can then decide whether to allow this specific request or not. This especially helps the user to identify network connections that are established in the background and are not triggered consciously by the user.

The dynamic Internet permission is particularly useful if the possible destinations of an application are not known in advance. The user can then enable the dynamic Internet permission for all outgoing connections and decide at runtime of the application whether a connection is permitted or not. We further allow the user to automatically generate new firewall rules from these dynamic request and store the decision permanently.

5.2.5 Observation of Internet connections

Ucicia also includes an *observation mode* that monitors and logs all outgoing network requests. The mode can be enabled for each application separately and currently logs the following information about connections:

1. The destination IP address,
2. the used network protocol,
3. the destination port, and
4. the HTTP host (if applicable).

The purpose of the observation mode is to make established connections of an application visible to the user. Therefore, the user can request a list of all logged connections.

[1] Further details about this dynamic Internet requests follow in the next section.

We further allow to create new rules from the logged Internet connections that either restrict or enhance the Internet permission of an application. This is especially useful because of the logged domain name that is otherwise not visible to the user. The domain name can give a more descriptive name that helps the user to identify the service an application is connecting to.

6 Implementation

As shown in Chapter 4, the overall goal of our extended permission model is to refine the network access in Android. To be precise, we allow to define a set of permitted IP addresses and domain names for each application and to deny the access to all other destinations. We enforce this refined Internet permission by utilizing the traditional Linux packet filtering firewall. For each outgoing packet, the firewall determines the application that wants to send the packet. Afterwards, it compares the destination IP address to the set of allowed hosts for this application and decides whether it let's the packet through or whether it drops the packet.

To configure the Linux firewall, we created an Android application, the *UciciaManager*. With this application, users can control the Internet access of each application by defining several rules that either allow or deny the communication to a specific destination. Further, users can postpone the decision to the time when the network access actually takes place. Therefore, the firewall queues the network packet and propagates all required information to the UciciaManager application. There, the user decides whether the connection is permitted or not.

We structured this chapter as follows: We first define the prerequisites of our implementation and provide an architectural overview. Thereafter, we take a closer look at the different modules. We start with the introduction of the used Linux firewall, then we present the interaction between the native firewall and the Android framework, and finally we describe the Android application to configure the system. Afterwards, we show how we further extended this implementation with a customized application installer and how we realized dynamic Internet requests.

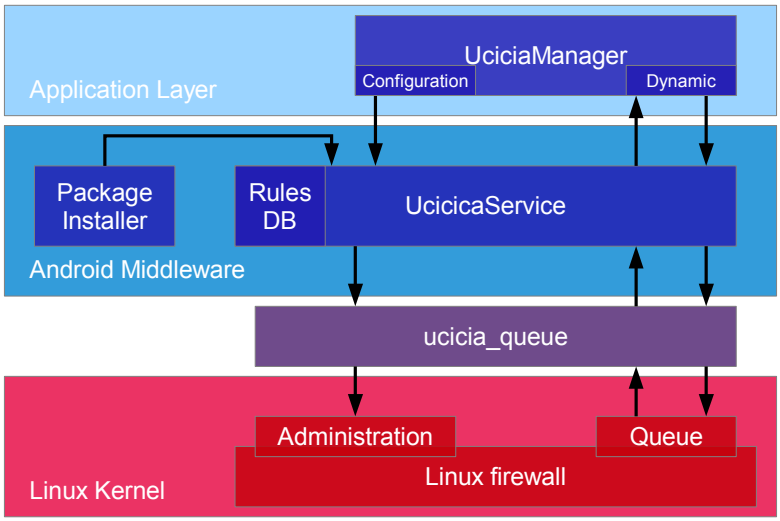


Figure 6.1: Interaction between the different modules. The left half illustrates firewall configuration, whereas the right half illustrates dynamic Internet requests.

6.1 Prerequisites

To implement our solution, we modified the stock Android firmware that is available through the AOSP. When we started our implementation, the latest available version of Android was Gingerbread (version 2.3.7). Nevertheless, we assume that our implementation can be ported to the new Ice Cream Sandwich release with reasonable effort.

We tested our implementation on Google's first developer phone, the Nexus One. The Nexus One, also known as HTC Passion, is shipped with an unlockable bootloader. Unlocking the bootloader allows to install unofficial firmware images and especially self-compiled firmwares retrieved from the AOSP repository.

The build configuration we used to compile the firmware was *full_passion-userdebug*. This configuration includes all available languages and applications (*full*), and provides root access and additional debugging tools (*userdebug*).

6.2 Architectural overview

The communication between the native Linux firewall and our Android application (UciciaManager) is realized by two system services. The *UciciaService* is located in the Android middleware and the *ucicia_queue* service runs as a standalone native application. The interaction between different modules of our implementation is illustrated in Figure 6.1. All modules in our implementation fulfill two different tasks: They are used to configure the firewall, and they inform the user about queued Internet connections.

The UciciaService stores all defined rules in an internal database and propagates them to the ucicia_queue service. There, the provided rules are used to update the state of the firewall. We give the user full control over the defined rules via the UciciaManager

application. Additionally, we give the application developer the possibility to ship a set of predefined rules required by the application. Therefore, we modified the *PackageInstaller* of the Android middleware, such that the provided rules are automatically added to the UciciaService at the time of the application installation.

The dynamic Internet requests are realized by queuing all applicable network packets and propagating them to the ucicia_queue service. There, we extract all relevant information (e. g. the destination IP address and the destination port) and forward this information to the UciciaManager. The decision of the user, whether the packet is accepted or dropped, is again propagated back to the ucicia_queue service. The service then informs the firewall about what should happen with the packet. The used term for this decision is *verdict*, which we use throughout this work to describe whether a packet is accepted or rejected.

6.3 Firewall

Each network packet sent from or to the Linux operating system passes Linux's internal firewall. The firewall distinguishes between incoming, outgoing, and forwarded packets. Incoming and outgoing packets are packets that are sent to and from the current machine, respectively. Whereas forwarded packets are just forwarded from one network interface of the machine to another network interface. The Linux firewall defines three different lists of filtering rules, so called *chains*: INPUT, OUTPUT, and FORWARD. Depending on the type of the network packet, the packet passes the appropriate chain. [51]

The rules inside these chains are checked one after the other against the inspected network packet. The filtering rules are, in general, based on properties of the IP, TCP, and UDP protocols. If one of the rules matches against the packet, the firewall jumps to the mandatory target that is set for each rule. There exist several predefined targets, but the most important ones are *ACCEPT*, *DROP*, and *REJECT*. All these three targets stop further processing of the packet and handle it in different ways:

1. The ACCEPT target hands the packet over to the operating system or to the end application.

2. The DROP target discards the packet. For the sender it seems like the packet never reached its destination.

3. The REJECT target also discards the packet, but additionally sends an error message to the sender of the packet. This error message informs the sender that the destination is unreachable.

In addition to the three predefined chains, the Linux firewall allows to create new chains. These chains can then be used as a new target for rules, such that the processing of the packet proceeds in the new chain. Figure 6.2 provides a short sample configuration that filters outgoing packets. The rules are checked from top to bottom, such that all connections to *192.168.12.1* are permitted and all network packets to *192.168.12.2* get dropped. If the destination IP address of some packet is *192.168.12.3*, we proceed to the

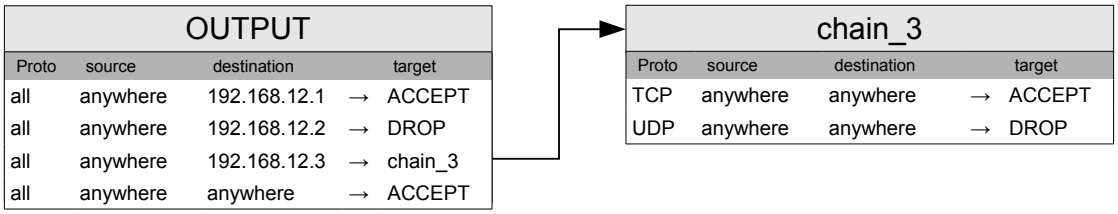


Figure 6.2: Sample firewall OUTPUT rules with one additional chain.

third rule and jump from there into the user chain *chain_3*. There, the protocol of the packet is checked. If TCP is used, we accept the packet, and if UDP is used, we drop it. In case the packet is neither a TCP nor a UDP packet, we jump back to the OUTPUT chain and proceed with the fourth rule.

6.3.1 Firewall configuration

The Linux firewall can be configured through the user space *iptables* program. This command line tool allows to create new and modify existing chains and filtering rules. The iptables program is part of the stock Android firmware that is available in the AOSP, but the available version is slightly outdated and did not fulfill our requirements. Therefore, we chose a newer version (1.4.7 instead of 1.3.7) from the well-known CyanogenMod[1] firmware.

The functionality of the iptables tool can be enhanced by several additional modules. These modules make it possible to filter packets based on further properties beyond the traditional protocol information. Our implementation makes use of the *owner*, the *NFQUEUE*, and the *mark* module. We will first introduce the owner module.

owner module The *owner* module is part of the iptables tool and allows to filter packets based on information of the process that sent a packet. The filtering only works for outgoing packets, because all other network packets obviously do not provide any information about the sending process. Outgoing packets can be filtered by several pieces of information of the Unix process that opened the corresponding network socket. This information includes the UID, the group identifier (GID), the process identifier (PID), and the session identifier (SID).

6.3.2 The chain design of Ucicia

As we discussed earlier, each Android application is executed with a distinct Unix UID. We utilize this information to distinguish different network packets, such that we are able to determine the application that tries to establish a network connection. This approach enables us to restrict the network communication of each application separately.

[1] CyanogenMod is a custom firmware that is based on the available Android source code on the AOSP. The firmware is quite popular because of its wide support for different devices. Further, it provides several features that are not available in the stock Android firmware.

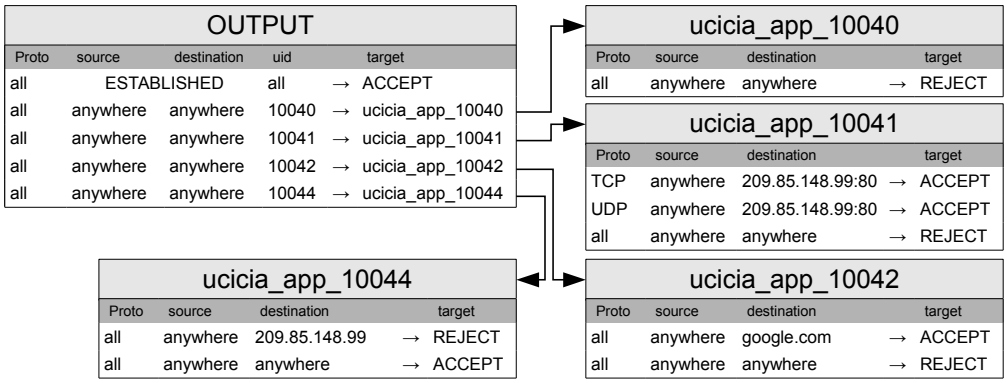


Figure 6.3: Ucicia firewall configuration restricting the Internet access of four applications.

To increase the performance of the Linux firewall, we add a special rule to the top of the output chain. This first rule in the chain uses the *state* module and makes use of the stateful filtering techniques of the Linux firewall. It matches all packets that belong to an already established connection and automatically accepts them without further processing any additional rule.

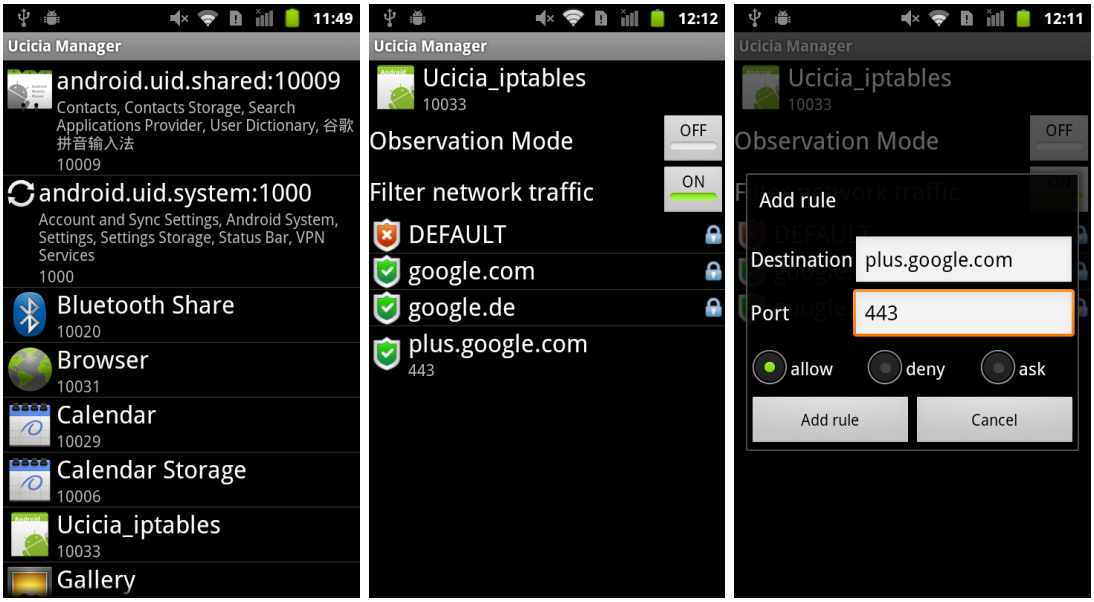
To further reduce the number of rules that are checked when a packet is sent, we cluster all rules that belong to the same application and place them into their own chain. The name of the new chain is determined by appending the application's Unix UID to the fixed prefix *ucicia_app_*. For each of these new chains, we place only one rule into the OUTPUT chain. This new rule in the output chain compares the UID of the packet with the UID of the application and in case they match, the rule jumps to the corresponding application chain. Figure 6.3 shows the linkage between the output chain and the separate *application chains*. As one can see, a packet send by the UID 10044 is only checked against the rules inside the OUTPUT chain and against the inside the ucicia_app_10044 chain and ignores all rules inside the other application chains.

The different types of filtering rules for an application are also depicted from Figure 6.3. Although, the Linux firewall supports a large variety of filtering properties, we concentrate on a limited subset. The target of all our filtering rules for an application is either ACCEPT, REJECT, or another chain. Further, we define three types of rules that are available in an application chain:

1. *Host-only rules.* These rules are based on the destination host address of a packet. The destination can either be an IP address or a domain name.

2. *Host-and-port rules.* In addition to the destination host, they define the destination port of a packet. For each such rule, we create two filtering rules, one for TCP and one for UDP.

3. *Default rules.* The last rule inside an application chain is always a rule that matches all packets. It states what will happen with the packets that are not handled by one of the previous rules.



(a) Main screen (b) Configuration screen (c) Add rule dialog

Figure 6.4: Screenshots of the user interface of the UciciaManager application.

These three rule types are sufficient to enforce the restricted Internet access as previously defined in Section 5.1. This can easily be seen by recalling the definition of restricted Internet access. We defined it as a set of restriction rules that consist of a host name (IP address or domain name) and a port. We can enforce such a set of restriction rules $R_{app} := \{(h_1, p_1), (h_2, p_2), \ldots, (h_n, p_n)\}$ for an application *app* with the UID u as follows:

1. We create a new application chain for the UID u,
2. for each tuple (h_i, p_i) we insert a corresponding host-and-port rule with an accepting target, and
3. we let the default rule reject all other packets.

6.4 Firewall front-end

We give the user the possibility to change the firewall rules through an Android application, the *UciciaManager*. The application itself is only responsible for representing the stored firewall rules and for modifying them, whereas the rules themselves are stored by the new system service that we present in the next section. Therefore, the architecture of the application is quite simple and only consists of three activities.

We added the application to the preinstalled applications of Android such that all the source code is located in the `/packages/apps/UciciaManager` directory of the Android repository. However, to distinguish our changes from the official source code, we did not adopt Android's package naming scheme. Instead we use the `de.unisb.infsec.ucicia`

package as the root package for all our changes. Therefore, the following three activities of the UciciaManager application reside in the `de.unisb.infsec.ucicia.manager.config` sub-package.

ListApplicationsActivity The main activity, that is started when the application is launched, is the `ListApplicationsActivity`. It lists all installed applications on the device that requested the Internet permission. To help the user to identify all applications, we present each application with its official name and its icon. We get the information about the installed applications, as well as their requested permissions, from the `PackageManager`, which is part of the Android framework.

Figure 6.4a shows a screenshot of this application list. As one can see there, several applications with the same Unix UID are grouped together. The reason for this design decision is that restricting the access for one of these applications will automatically influence the permission of all other applications with the same UID. Therefore, we decided to list these applications with the same UID as only one item. We assume that a group of these applications possesses the Internet permission as soon as at least one application requested it.

We use this application list to select the application we want to configure. A click on the application brings us to the configuration screen for the selected application.

ConfigApplicationActivity The user can control the Internet access of each application through the `ConfigApplicationActivity`. At the top of the screen, we again provide the essential information about the application that is being modified. Below, we placed the controls to represent and modify the current configuration. We receive the current configuration of the application from a new system service that we added to the Android framework. Further details about this service follow in Section 6.5.2.

As depicted in Figure 6.4b, the user can switch the packet filtering either on or off, where off falls back to the stock Internet permission and allows full Internet access. If the user decides to activate the packet filtering mode, we present the list of restrictions that are currently stored for this application. Each listed rule item contains the relevant information, which is

1. the destination host (located in the first row),
2. the destination port (located in the second row), if applicable,
3. the mode of the rule represented by an icon on the left (where a green tick indicates an accepted, and a red cross a denied connection, respectively), and
4. the state of the rule that is either built-in (indicated by a small lock icon on the right) or normal (where no special icon is presented).

The first rule in the list is a special built-in rule, the *DEFAULT* rule. It is the equivalent to the catch-all rule we have seen when we talked about the firewall design. Thus, it represents the default behavior for all packets that do not match any of the

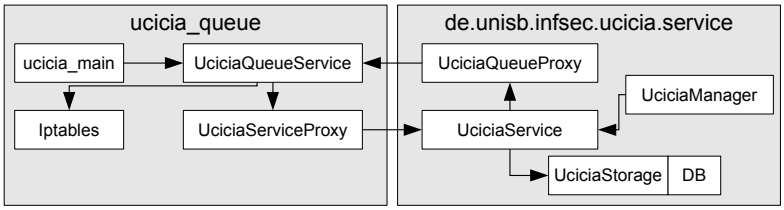


Figure 6.5: Communication between the different modules of the two Ucicia services.

other rules. The default rule can be used to block the network access of the application by setting the mode to *deny*.

We allow users to delete any user-created rule. However, built-in rules can not be deleted such that the user can always easily restore the initial state of the application. Further, users can change the mode of all rules (including the built-in rules) and decide whether to accept a particular connection or to deny it.

New rules can be created through the Android application menu. We handle the creation of new rules in a separate activity.

AddRuleActivity The user interface of the `AddRuleActivity` is depicted in Figure 6.4c. It provides a simple interface that asks the user for the destination host, the port, and the mode of the new rule. The provided information is used afterwards to create a new rule via the new `UciciaService` system service.

6.5 Framework interaction

The architectural design of the framework interaction was influenced by the fact that some parts of the implementation require root privileges to fulfill their task. Unfortunately, there is no way to run an Android application itself as root. Therefore it was necessary to implement a native application that is not started by the Android middleware. In return, such native applications can not directly interact with the user through a UI.

To solve this contradiction, we created two *system services*: One in the native application (ucicia_queue) and one in the Android middleware (UciciaService). The two services communicate through Android's Binder IPC mechanism. *System services* should not be confused with the `Service` application component that we presented in the chapter about application development. System services only implement the Binder interface and announce this interface to a special Android component, the *ServiceManager*. Other components can contact this service manager to retrieve the Binder interface by its unique name.

Due to these two services, we can now establish a bidirectional communication between the native application and the Android middleware. Figure 6.5 outlines the different modules of the two services and their communication. We protected the communication between the services by introducing a new permissions (`UCICIA_QUEUE`). We check the

permission explicitly for all calls that arrive through the Binder interface and deny the access in case the caller does not possess this permission. The protection level of the permission is set to *system* such that no third-party application is able to interact with the services in an unintended manner. In the following, we will have a detailed look at the native application and the service in the Android middleware.

6.5.1 The native `ucicia_queue` application

The main task of the `ucicia_queue` application is to receive firewall configurations from the UciciaService and to configure the native Linux firewall accordingly. The application is located in the `/external/ucicia_queue/` directory of the Android repository. The programming language used for the implementation is C++. We choose C++ mainly because of the available Binder library that provides several templates to easily create a new Binder interface.

Our modifications heavily rely on this new service, therefore we have to ensure that it is always running. To start applications automatically, Android uses the `init.rc` file that is located in `system/core/rootdir/`. This file allows to start the service automatically at boot time and it further restarts the service in case it may crash. The autostart mechanism is triggered by the following new line in the `init.rc` file:

```
service ucicia /system/bin/ucicia_queue
```

To keep the complexity of the application rather low, we structured it into four modules: The main module, the service module, the firewall module, and the proxy module.

The main module (`ucicia_main`) The main module contains the `main` method of the application, which is called when the application gets started. The only purpose of this module is to initialize and start the service module.

The service module (`UciciaQueueService`) The service module contains the code to create a new Binder interface. It consists of several helper classes that are required to register the `UciciaQueueService` as a system service. New system services are registered by calling the `addService` method of the default service manager. The methods expects an instance of the Binder interface (i. e. the UciciaQueueService object) and an unique name for this service (in this case "*ucicia_queue*").

All incoming calls to the UciciaQueueService are handled in its `onTransact` method. The input to the method is an integer number (*code*) and a `Parcel` object. The code distinguishes between different operations provided by the service. The onTransact method of the UciciaQueueService allows to update the firewall configuration of a specific application.

All required information to update the firewall is provided in the lightweight Parcel object. This especially includes the UID of the application and several filtering rules. After extracting this information, the service module updates the firewall by calling the appropriate methods of the firewall module.

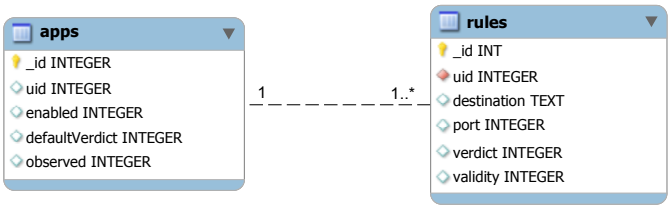


Figure 6.6: Database model used by the storage module.

The firewall module (`Iptables`) The Iptables class provides several methods to change the state of the Linux firewall. It abstracts from the different configuration switches of the iptables tool by providing specialized methods. The methods transform the given arguments into the appropriate iptables commands.

We use the native `system()` call to execute the appropriate iptables commands. Calls to the iptables binary require root privileges. Therefore, the ucicia_queue application has to be started by the root user. A list of all calls to the iptables tool is available in Appendix A.

The proxy module (`UciciaServiceProxy`) We created the simple UciciaServiceProxy class to access the `UciciaService` in the Android middleware. It provides the same methods as the middleware service and handles all the IPC tasks. Therefore, the class requests the Binder object of the second service from the service manager and encapsulates all given arguments in a `Parcel` object[2] that is afterwards transmitted to the service.

6.5.2 The `UciciaService` in the Android middleware

The `UciciaService` is the counterpart to the ucicia_queue service and resides in the Android middleware. Together with the ucicia_queue service, it is the link between the native Linux firewall and the Android framework and is responsible for the management of all filtering rules. The source code of the UciciaService is separated into two modules of the Android framework: The core functionality of the service itself is part of Android's *service module*[3] and the functionality that allows to interact with the service through the API is part of Android's *core module*[4].

The service is divided into several modules: A storage module to store the configurations, a proxy module to communicate with the ucicia_queue service, the Binder service itself, and a manager module that allows other Android components to easily communicate with the service.

The storage module (`UciciaStorage`) All defined rules are stored and managed by the UciciaService. To store the different configurations of all installed applications, the

[2] A Parcel object is a lightweight object that allows to transmit a sequence of scalar values between processes.
[3] Located in the `frameworks/base/services/` directory.
[4] Located in the `frameworks/base/core/` directory.

UciciaService makes use of a SQLite database. We abstracted from the underlying storage technique by creating the `UciciaStorage` class that provides standardized methods to store and retrieve such configurations.

We store all data in two different database tables; their database model is illustrated in Figure 6.6. The *apps* table stores the configuration of each application and the firewall rules linked to an application are stored in the *rules* table. Each of these tables contains a unique key that is stored in a separate *_id* column. Further, each table contains a *uid* column. In this column, we store the applications UID, so that we are able to link configurations and corresponding applications.

To exchange stored data, we created several lightweight data transfer objects: *AppSetting*, *UciciaRule*, *Validity*, and *Verdict*. The AppSetting class and the UciciaRule class represent a row of the apps table and of the rules table, respectively. The *Validity* enumeration is a Java enum that describes the state of the filtering rule, which is either user-defined (`Validity.ALWAYS`) or built-in (`Validity.BUILDIN`). Additionally, the *Verdict* enum describes the type of the rule and determines whether a connection is allowed (`Verdict.ALLOW`) or not (`Verdict.DENY`). The name of the *Verdict* enum corresponds to the verdict that the Linux firewall uses to decide on an inspected network packet (e. g. ACCEPT or REJECT).

The proxy module (`UciciaQueueProxy`) We again created a lightweight proxy module that provides the same methods as the ucicia_queue service and handles the necessary Binder communication.

The service module (`UciciaService`) The service module contains the main functionality and the programming logic of the Android service. It uses the storage module to store all configurations persistently and updates the state of the Linux firewall via the proxy module. The firewall front-end application accesses this module to read and update the stored filtering rules of the applications. The communication between this service and the Android application is again realized via the Binder interface.

Since the UciciaService is part of the Android framework, we made use of the Android Interface Definition Language (AIDL) to define the Binder interface of the service module. Therefore, Android's build system translates the `IUciciaService.aidl` file in the `frameworks/base/core/java/` directory into a Java stub class. The data exchange is again realized by using our custom AppSetting, UciciaRule, Validity, and Verdict structures. To exchange them via the Binder interface, all of them have to implement the `Parcelable` Java interface and we transmit these objects by marshaling their information into scalar types.

Like several other core Android services, our service is started by Android's system server. The `com.android.SystemServer` initializes the UciciaService and adds its instance to the `ServiceManager`, such that the service is available via its unique name (i. e. "*ucicia_service*"). Consequently, the UciciaService is part of Android's system process.

All changes to the Linux firewall only persist as long as the device is running and all rules are lost as soon as the device is rebooted. Therefore, the service module contains

a broadcast receiver that waits for the system event that is sent after the boot process finished (`Intent.ACTION_BOOT_COMPLETED`). As soon as the boot process finished, we configure the Linux firewall and add all necessary rules to restrict the Internet access.

The manager module (`UciciaServiceManager`) The Android framework contains *system services* that are available to all applications via the Android API. For each of those system services exists a proxy object that communicates with the actual service through the Binder interface. Thereby, the proxy object provides an API to easily access the service.

We implemented the UciciaService as a system service, such that the UciciaManager application can easily interact with the new service. The `Context` class of the Android framework allows to retrieve a handle to such a system service via the `getSystemService` method. We modified the implementation of this method in the `ContextImpl` class and added a new proxy object, the `UciciaServiceManager`, that communicates with the UciciaService.

The manager module consists of the UciciaServiceManager, several helper classes, and several AIDL interfaces. Since the manager is available to all applications, its source code is located in the `frameworks/base/core/java/` directory. The UciciaServiceManager is available to all applications that run on the device since it is part of the *core* Android module. Therefore, the new service leads to a change in the official Android API[5].

6.6 Application Package Installation

In addition to the manual firewall configuration that we provided in the last two sections, we modified the application development and the application installation process. Instead of requesting full Internet access, our modifications allow a developer to request fine-grained Internet permission. Further, users can install applications, that request the Internet permission, without granting them the requested Internet permission.

6.6.1 Rules definition in the `AndroidManifest.xml` file

It was of great interest for our solution to involve the developers into the definition of the appropriate firewall rules, because they have more technical background and they can better decide which Internet connections are actually mandatory to run the application.

We therefore enhanced the format of the `AndroidManifest.xml` file with respect to the permission requests. We decided to add a new *restriction* attribute to the `uses-permission` tag in order to make the Internet permission request more fine-grained. In order to request full Internet access, developers can still request the Internet permission without using any restriction; but they can also limit the request to a limited subset of destinations by requesting fine-grained access to each destination separately. Figure 6.7 compares the two approaches and provides an example for a restricted Internet permission request.

[5]Consequently, a `make update-api` is required to build the new firmware.

```
<uses-permission android:name="android.permission.INTERNET" />
```

(a) Full Internet access.

```
<uses-permission android:name="android.permission.INTERNET"
        android:restriction="google.de" />
<uses-permission android:name="android.permission.INTERNET"
        android:restriction="google.com" />
<uses-permission android:name="android.permission.INTERNET"
        android:restriction="g.cdn.com" />
```

(b) Restricted Internet access.

Figure 6.7: Comparison between request of full and restricted Internet permission.

The format of the Android manifest file is defined in the `attrs_manifest.xml` file in the `frameworks/base/core/res/res/values` directory. We note that a modification of this file implies an update of the Android API. Such API changes have to be explicitly confirmed when compiling the new firmware[6], since they may introduce incompatibilities with older devices. Nevertheless, we assume that our API change is rather small and should not hinder our solution from being included in the official Android source code.

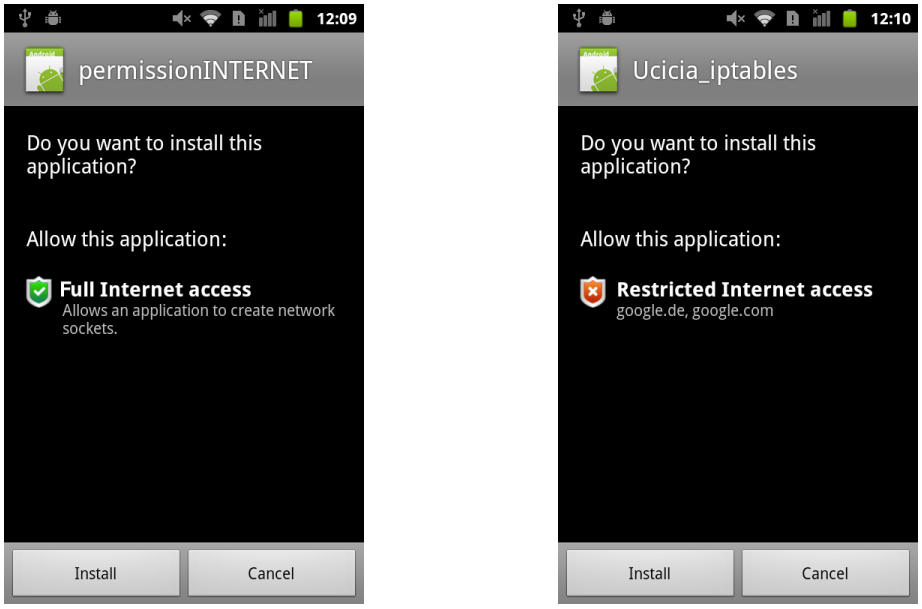
To read the new attribute, we modified the `PackageParser` class and store the list of defined restrictions for each permission. We use this information when the application is actually installed on the device.

The `PackageManagerService` is responsible for the actual installation of the application. We extended this framework service such that our `UciciaService` is notified about new applications and their defined restrictions at the end of the installation process. This allows us to create the appropriate firewall rules if the application requested the Internet permission. In case the developer requested restricted Internet access only, we create one *accepting* filtering rule for each provided restriction and deny the access to all other destinations by setting the default target to *deny*. We mark all extracted rules as `BUILDIN`, such that we can distinguish them from user generated rules. We later especially use this distinction to inform the user about which rules have been defined by the developer. However, if the developer did not specify any restriction and therefore requested full Internet access, we do not enable the firewall for this application and don't add any new rules.

6.6.2 Revocation of the Internet permission

In order to give the user more control over the Internet connections on the device, we modified the stock application installer of Android. The default installer lists all requested permissions of an application and marks dangerous permissions with an additional icon. The user should use this information to decide whether to install the application or not.

[6] One has to run `make update-api` in order to successfully compile the new firmware.



(a) Full Internet access (granted) (b) Restricted Internet access (revoked)

Figure 6.8: Modified package installer showing full and restricted Internet access.

Android permissions are grouped into different categories. Consequently, the Internet permission is grouped together with several other network related permissions. We modified the default grouping behavior of Android for the Internet permission and always show this permission in a separate category. All restrictions provided by the developer are then grouped in this new category. We think that this grouping allows the user to easily retrieve the desired information about the requested Internet permission. Figure 6.8 shows two screenshots of the modified package installer: One shows a request of full Internet permission and the other one shows a request of restricted Internet permission.

We modified the UI of the installation process and give the user the possibility to revoke the requested Internet permission. With a click on the Internet permission, the user can toggle between *granted* and *revoked*. Depending on the user's choice, the post-installation process generates appropriate firewall rules that either allow or deny Internet access for an application.

There are several different components of the Android framework that are involved in the installation process of an application: First of all, all stock permissions of the Android framework are defined in a specific `AndroidManifest.xml` file; secondly, the `PackageInstaller` is called when a new application is being installed on the device; thirdly, the package installer uses the `AppSecurityPermissions` class to generate the view that represents all requested permissions of an application; and finally, the `PackageManagerService` performs the actual installation of an application.

In order to handle the Internet permission separately, we added a new permission

group, named `RESTRICTED_INTERNET`[7], to the `AndroidManifest.xml` file of the Android framework[8]. Further, we modified the `AppSecurityPermissions` class to generate a customized permission list. We manually change the group of the Internet permission from the official `NETWORK` group to our new `RESTRICTED_INTERNET` group. This allows us to emphasize the Internet permission by showing it in its own category and to list all defined restrictions as part of the permission view, as seen in Figure 6.8b. To revoke the requested Internet permission, we allow users to click on the Internet permission, which toggles between a *granted* and a *revoked* Internet permission.

The `PackageInstaller` application handles the installation of new applications and guides the user through the installation process. The main duty of the package installer is to let the user confirm requested permissions of an application explicitly. Therefore, it receives the appropriate permission view from the `AppSecurityPermissions` class and asks the user whether to proceed with or cancel the installation. If the user confirms the installation, the package installer triggers the installation of the application by calling the `PackageManagerService`. We only had to slightly modify the package installer, such that the information about the Internet permission revocation is also transmitted to the package manager.

In addition to the restrictions we extract from the `AndroidManifest.xml`, we also pass the user's decision, whether to grant the requested Internet permission or not, to the `PackageManagerService`. If the user grants the Internet permission to the application, the Linux firewall is configured as before. However, in case the permission request is revoked, the target of all filtering rules is changed to *deny*. This includes the developer-defined rules as well as the default rule with the result that all network access of the application is blocked.

6.7 Dynamic Internet permission

In addition to the static Internet permission that is granted before the permission is actually needed, we wanted to make the Internet permission more dynamic. Therefore, we further enhanced our solution and give users the possibility to postpone the decision as to whether a network connection is allowed or not, to the time when the connection is actually established.

This *dynamic Internet permission* requires several modifications related to our solution and to the stock Android firmware. We reconfigured the Linux firewall, such that we are able to queue all network packets. Further, we let the ucicia_queue service inspect all those network packets and extract information about the connection and about the application that wants to establish the connection. The identification of the source application of a network packet required slight modifications of the Linux kernel. Finally, the extracted information about the network connection is forwarded to the UciciaManager application that asks a user whether the connection is allowed or not.

[7]The full group name is `de.unisb.infsec.ucicia.perm-group.RESTRICTED_INTERNET`.
[8]The file is located in the `frameworks/base/core/res` directory of the repository.

6.7.1 Firewall modifications

To implement the dynamic filtering approach, we needed to intercept outgoing network packets, to add them to a queue, and to ask the user if the connection is allowed or not. Usually, the firewall filtering is handled by the Linux kernel according to the defined filtering rules. However, the Linux firewall allows to transfer network packets from the kernel to the user space. This can be achieved by an additional firewall module, the NFQUEUE module.

NFQUEUE module The NFQUEUE module adds a new rule target to the Linux firewall. In addition to the default ACCEPT and REJECT target, we can set the target of a filtering rule to *NFQUEUE*. All network packets that match a filtering rule with this target are added to a queue and can be further processed by a user space program. The network packets can be added to several different queues by setting an optional 16-bit *queue number*. If no queue number is specified for a rule, the packets is handled by the default queue with the queue number 0. The user space program can afterwards listen to one or several of those queues.

In order to create such filtering rules, we recompiled the `iptables` program with the enabled NFQUEUE module. The module enhances the supported arguments of the `iptables` program, such that the target switch supports the `NFQUEUE` target, and an additional `queue-num` switch is available to set the desired queue for the filtering rule (e. g. `iptables -j NFQUEUE --queue-num 42`).

6.7.2 Kernel modifications

The NFQUEUE module allows to inspect the content of each queued network packet (e. g. the destination IP address or the source port) in a user space application. However, the owner information about a packet is not available when handling packets in the user space. In order to provide meaningful information about the queued connection to the user, we need to determine the application that wants to establish the Internet connection. Therefore, we need to know the UID of the application that wants to send a packet.

To retrieve this information in the user space program, we had to slightly modify the `nfnetlink_queue` module of the Linux kernel. The appropriate kernel for the Google Nexus One device is available under the name *msm*. Unfortunately, the official Android repository on kernel.org was not available[9] when we tried to get the kernel source code. Therefore, the Kernel source code was retrieved from an unofficial mirror on github[10]. The latest available version of the Kernel (version 2.6.35.7-59465) was used to implement the required modifications.

We modified the `net/netfilter/nfnetlink_queue.c` file that is part of the Linux kernel and extract the UID from the file handle of the network socket. The extracted UID is stored together with the other information about the network packet. To retrieve the stored

[9] The Linux Kernel Archives was offline due to an hacking attack.
[10] https://github.com/android/kernel_msm

UID, we further modified the corresponding `include/linux/netfilter/nfnetlink_queue.h` header file and added a new attribute that identifies this new property of a network packet.

6.7.3 ucicia_queue modifications

We extended the functionality of our ucicia_queue service and added a new module, the *UciciaQueue* module. This module registers itself to handle all network packets of the queue with the queue number 1. For each packet, we extract the most important information (e. g. the destination IP address, the destination port, and the Unix UID) and forward it to the UciciaService in the framework. After a user made the decision as to whether an connection is allowed or not, we tell the Linux firewall the *verdict* for the network packet (which is either ACCEPT or REJECT).

In order to communicate with the Linux firewall, we included two additional libraries into the Android firmware. We need them to develop the UciciaQueue module.

libnetfilter_queue The *libnetfilter_queue* library provides an API to create new user space programs that listen to queued packets of the Linux firewall. We added the latest version (i. e. version 1.0.0) of the library to the source tree of the Android operating system and compiled it as a shared system library.

Further, we slightly modified the library and added a new function that returns the UID of the application that tried to send the queued network packet.

libnfnetlink The libnetfilter_queue library requires the *libnfnetlink* library. Therefore, we added the latest version (i. e. version 1.0.0) of the library to the Android firmware. The library provides the required low-level functionality for netfilter related communication between kernel and user space.

UciciaQueue module

The ucicia_main module initializes the UciciaQueue module at startup and afterwards executes the module in a separate thread. The additional thread enables us to handle incoming network packets and request from the framework service simultaneously. The UciciaQueue module uses the libnfnetlink_queue library to receive all queued network packets for the queue number 1. For every packet, the `handle_packet` callback function is called. There, we extract the following information for each queued packet:

1. The UID of the application that has sent the packet,
2. the unique ID of the packet,
3. the destination IP address,
4. the used transport protocol (e. g. TCP or UDP), and
5. the destination port (for TCP and UDP connections only).

We transmit this data to the framework service, such that the user can decide whether the queued Internet connection is accepted or not. As soon as the user made the decision, we call the `nfq_set_verdict` function of the queue library. This function determines whether the packet is accepted or dropped. To identify the corresponding network packet, the method expects the unique ID of the packet in addition to the verdict argument.

While the application waits for the user's decision, it may happen that more packets with the same destination (i. e. the same UID, IP address, and port) arrive. To limit the communication overhead between the native service and the framework, we only send one notification per distinct connection. For all similar packets, we only store their unique ID and process them as soon as the user decided about the connection.

To store the information about the packets, we created a new `Connection` class that stores the UID, the IP address, and the port together with the unique IDs of all corresponding network packets. We further created two lookup maps to easily retrieve those connection objects from a given $(uid, ip, port)$ tuple and from a given packet ID: We use the tuple map whenever a new packet is queued and check whether we already notified the user about this specific connection. If the packet belongs to a new connection, we notify the user and otherwise, we store the unique ID of the packet together with the known connection. Whenever the user sends the decision about a queued connection, the ID map is used to retrieve all collected packets that belong to the same connection. Afterwards, we set the verdict of all those packets and let the Linux firewall either ACCEPT or REJECT those packets.

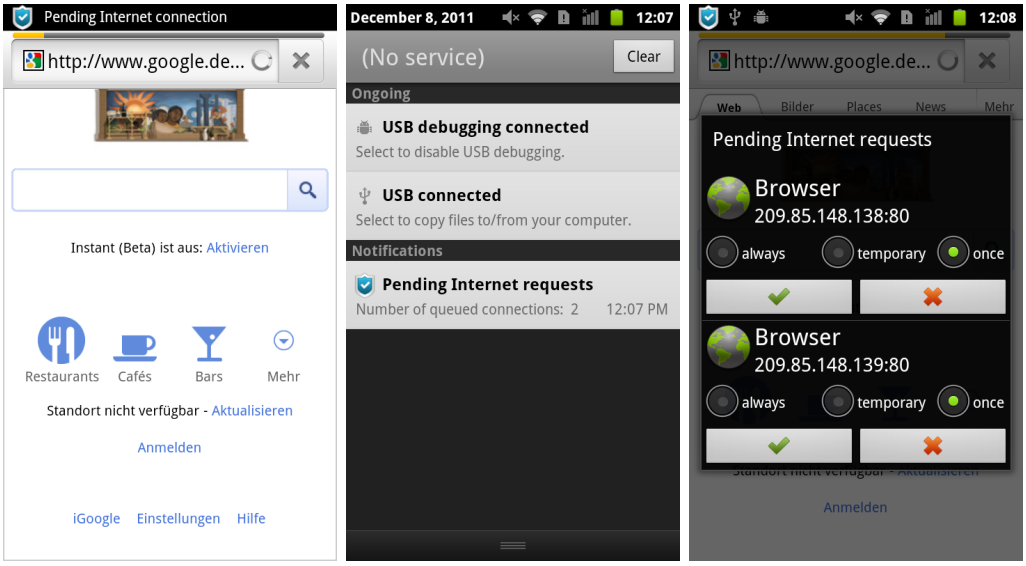
We further use the connection object to temporarily save the decision of the user. This allows us to reuse the decision in a predefined time window without asking the user again. We therefore choose a time window of 30 seconds and assume that this does not weaken the security of our approach, since the connection has been established anyway.

6.7.4 UciciaService and UciciaManager modifications

The standard way of notifying the user on Android is to use the notification system. Any application can send its messages to the notification service and the notifications are placed into the status bar at the top of the screen. The user can then click on the notification to handle the event. The notification system allows to gain the user's attention without interrupting the current task.

Therefore, we also use this notification system to inform the user about pending Internet requests. As soon as the user decides to handle the requests, we present a list of all queued Internet connections and allow to accept or deny them through a simple UI. In order to show the notifications, we forwards all requests that arrive in the UciciaService to the UciciaManager application and trigger the notification there. Further, we created a new activity, the `VerdictActivity`, that provides the UI to accept and deny the pending Internet requests.

The communication between the native ucicia_queue service, the UciciaService, and the UciciaManager application is handled as follows: Firstly, the UciciaService provides a new Binder method (i. e. `confirmRequest`) that is called by the ucicia_queue service and that transmits all information about the queued Internet connection to the Android



(a) Notification bar (b) Notification screen (c) VerdictActivity

Figure 6.9: Screenshots of the dynamic Internet permission notification and of the user interface to allow and deny a queued Internet connection.

framework. Secondly, the UciciaService accepts so called `UciciaListeners` that get notified whenever a new request arrives; the UciciaManager application is such a listener. Thirdly, the UciciaService provides a second Binder method (i. e. `verdictRequest`) that is called by the UciciaManager. This call includes the user's decision about the Internet connection and is propagated to the native ucicia_queue service and consequently to the Linux firewall.

To make the UciciaManager application capable of receiving requests in the background, we added a new service component, the `NotificationService`, to the application. We start the service directly after the device booted successfully. The service implements the `UciciaListener` interface and registers itself as a listener of the UciciaService whenever the notification service is started. Consequently, the NotificationService is notified about all queued Internet requests. Whenever the notification service receives a request, it adds a notification to the notification service. Figure 6.9a and 6.9b show the status and the notification bar for this notification.

If the user clicks on the notification, the new `VerdictActivity` is started. It requests the list of all pending requests from the UciciaService and provides the UI to handle each request separately. Figure 6.9c shows the interface of the activity. The look of the activity mimics a default dialog, such that the transparent background lets users see the previous action. For each request, the activity provides one *accept* and one *deny* button. Further, users can choose how long the decision stays valid. Either an connection is accepted *once*, *temporary*, or *always*; where the later two automatically create a filtering rule in the Linux firewall that accepts connections with the same UID, IP address, and port. However, the temporary rule only stays valid until the next reboot of the device.

6.8 Observation mode

The definition of appropriate filtering rules is quite difficult if one tries to limit the Internet access of an unknown application. We therefore added an *observation mode* that logs and reveals all Internet requests of an application. After enabling this mode, users can retrieve a list of all logged Internet requests of an application. Consequently, users can check this list to decide whether to trust the application or not. In addition, we give users the possibility to create appropriate rules from the logged requests.

The required modifications are quite similar to those of the dynamic Internet permission in the last section. We again use the queuing possibility of the Linux firewall to inspect the desired network packets and extract the important information about the connection. The gathered information is afterwards forwarded to our service in the Android middleware and finally stored in a database. We further had to change the design of the firewall rules to ensure that we definitely log all desired network packets.

6.8.1 Firewall modifications

To log each network packet of an application, we have to route it through our user space program before it is finally accepted or rejected by the firewall. Therefore, we add a new filtering rule at the top of the output chain before any other rule may decide on the network packet. The rule queues the packet such that we can process it in the ucicia_queue application. After extracting all required information, we want to handle the packet as before. In order to do that, we reinsert the packet again into the firewall.

Unfortunately, a queued packet can not be insert right after the chain that queued it; it can only traverse the whole output chain again from the beginning. However, this can lead to infinite loops, if the packet is reinserted again and again. The libnetfilter_queue library allows to mark a reinserted packet with an integer number. We use this mark in the observation rule such that we queue the packet only in the first iteration and proceed with the other rules in the second run.

mark module The *mark* module allows to check the mark of a network packets. For any filtering rule in the output chain, one can use the `--mark` switch to check the assigned mark of the packet. The default mark value of each packet is 0, but it can be changed to an arbitrary 32-bit integer value by the `verdict2` function of the libnetfilter_queue library, when a packet is reinserted into the firewall.

Modified chain design of the firewall To support the observation mode, we had to modify the chain design of the firewall. We created a distinct chain that contains all rules related to the observation mode. The name of the new chain is `ucicia_app_observe`. In order to inspect each network packet, we added a new rule to the top of the output chain. The rule checks the mark of the network packet and proceeds with the new `ucicia_app_observe` chain if the mark is unchanged and hence still set to 0. In all other cases, we proceed with the next rule in the output chain.

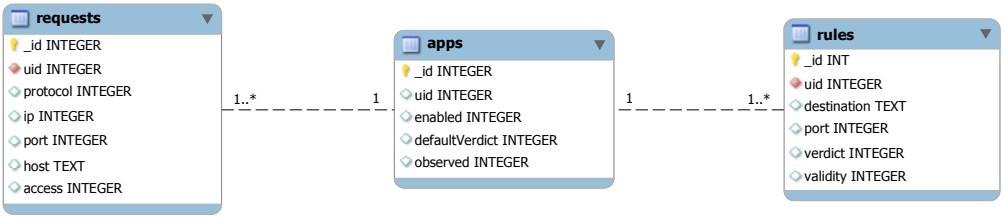


Figure 6.10: Enhanced database model with an additional table to store all observed requests.

If the observation mode of an application is enabled, a new rule is added to the `ucicia_app_observe` chain. This rule checks the UID of the packet and forwards all packets with the same UID as the application into a distinct queue (i. e. with the queue number 2).

We assume that the number of applications with an enabled observation mode is low, such that it is no problem that each packet passes the `ucicia_app_observe` chain, since it does only contain a limited number of additional rules.

6.8.2 ucicia_queue modifications

All observed packets are stored in queue number 2, so we added a second callback function `observed_packet` in the native ucicia_queue application that receives all packets of this queue. Like for the dynamic Internet permission, we extract all the important information about the connection (i. e. the UID, the protocol, the IP address, and the port) out of the packet, but we also try to extract the HTTP host address.

In general, a HTTP request is send over TCP on port 80; therefore, we extract the content of each such network packet and try to extract the `Host` field from the HTTP header. For all HTTP requests to a domain name, the host field contains the domain name that is otherwise not available after the DNS resolution. After we extracted all the important information, we send it to the UciciaService.

For all the observed packets, we don't have to wait for any user decision, such that we can immediately reinsert the packet into the firewall. We do so via the `verdict2` function of the libnetfilter_queue library and the `NF_REPEAT` target. We additionally mark the packet as being already processed by setting its mark to the value 1. This ensures, that the packet does not enter the `ucicia_app_observe` chain again.

6.8.3 Database model modifications

In order to log all the observed requests of an application, we modified the database model and added a new *requests* table. Figure 6.10 shows that the new table contains several columns to store all the available information about a network connection. This includes network related information like the transport layer protocol, the destination IP address, the destination port, and the HTTP host name, but also the UID to identify the application, and the date and time of the access.

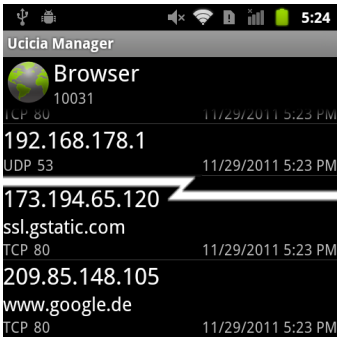


Figure 6.11: Observed requests of the Browser application.

6.8.4 UciciaService and UciciaManager modifications

To receive the information about an application, we added a new Binder method that allows the native ucicia_queue service to transmit the information to the framework. This enables us to easily store all the observed requests in the database. Further, we added an additional Binder method that allows to retrieve a list of all observed Internet connections for a specific application.

The `ObservedRequestsActivity` of the UciciaManager uses this method to present the gathered information to the user. It list all the connections chronologically and gives the user the possibility to check which connections an application tries to establish. Each of the item in the list can be used to create a new firewall rule based on the provided information. This helps the user to create new rules based on the domain name instead of the IP address that is normally not visible to the user. Figure 6.11 provides an excerpt of such a connection list.

7 Performance Evaluation

We measured the performance of our implementation in order to estimate whether user-controller Internet connections are feasible on Android devices. For our estimation, we concentrated on the network performance only and did not measure any other performance criteria, like CPU load, memory consumption, or energy consumption.

Therefore, we measured the network throughput of our solution with several scenarios and compared the results to the standard behavior of Android. We expected that the overhead of our implementation is low if the network packets are directly handled by the Linux firewall and are not handled in the Android framework. In these cases, the overhead of our solution should correlate with the overhead by the Linux firewall and we expect that the firewall can efficiently handle the network traffic on the phone.

For the dynamic Internet approach, packets are passed to the user space application and the dynamic request also involves interaction with the framework. However, we kept the framework interaction to a minimum and only conduct one request for each distinct connection. After the user decided on the connection, further network packets are handled by the native user space application only. We expect that the user space application should not slow down the network throughput tremendously.

7.1 Methods and tools

In general, smartphones support two types of network connections: Connections over Wi-Fi and connections via mobile broadband. We analyzed the network throughput of the Wi-Fi network in our experiment. The reasons for choosing the Wi-Fi network are quite obvious: Firstly, the Wi-Fi network is in general faster than mobile broadband connections and therefore does not limit the connection in advance. Secondly, we have more control over Wi-Fi connections, since we can define the used hardware and influence the quality of the connection.

In our experiments, we measured the *average network throughput* (or in short *average throughput*). The average throughput states how much data can be transmitted in a specific time. Assuming we commit x bits and the transfer takes t seconds, the average throughput is x/t bit/sec [38]. It is worth mentioning, that a kilobit consists of 1000 bits instead of 1024 bits. Consequently, a throughput of 1 Mbit/s (or 1 Mbps) refers to $1,000,000$ bit/sec.

In order to measure the network throughput, we had to set up the required hardware and software. In the following, we will describe how we connected the different hardware devices and which software tools we used for the performance measurement.

7.1.1 Hardware setup

In order to measure the network throughput, we had to send data from the phone to another location in the network. The other location in the network needs to be capable of measuring the number of received bits in a specific time. Further, we have to ensure that the Wi-Fi communication of the phone is the slowest link in the connection chain. This demand leads to the following test set-up for our performance measurement. It consisted of three devices:

1. The Nexus One developer phone (with our custom firmware),
2. a wireless router (Speedport W 701V, Deutsche Telekom AG), and
3. an average notebook.

We connected the Nexus One via a *802.11g* Wi-Fi connection to the router. The 802.11g transmits up to 54 Mbit/s of raw data. However, the actual average throughput may be lower because of the overhead of the used protocols (e. g. TCP).

In order to avoid that the connection from the router to the notebook is the bottle neck in the communication, we connected the notebook via a cabled Ethernet connection to the router. We therefore used a 100 Mbit/s (100BASE-TX) connection. The data rate of this connection is almost twice as big as the wireless communication and should therefore not slow down the speed of the communication between the phone and the notebook.

7.1.2 Applied software

In order to determine the average throughput of the smartphone, we had to send data from the phone to another location in the same network and measure the number of transmitted bits in a specific time window.

We used several tools that help us to set up the connection between the phone and the network target and to measure the number of transmitted bits per seconds.

Wifi Analyzer

We have to make sure that the wireless network connection does not influence the result of our measurement. Therefore, we measured the signal strength of the device, in order

to position the device and the router in an opportune position. A high signal strength ensures that the throughput of the wireless connection is as high as possible.

The *Wifi Analyzer* application is freely available in the Android Market and contains several tools to analyze the Wi-Fi environment. We used version 2.5.16 of the application to measure the signal strength of the router that the phone was connected to.

We arranged the router and the phone in such a way that we get the highest possible signal strength between these two devices. We achieved a signal strength of about -20 dBm, which was the highest value the application could represent. We do not treat this value as absolutely correct, however we assume that the represented strength is a good indicator to keep the signal strength as high as possible.

Iperf

We measured the average network throughput of the phone with the *iperf* application[1]. Iperf can be used to measure the performance of TCP and UDP connections.

In order to measure the network throughput between two applications, the iperf application has to run on both machines. One machine runs as the server and the other machine connects to this server as a client. After the client connected to the server, the client starts sending data to the server for a specified interval. At the end of the measurement, the server calculates the average network throughput.

We run the iperf tool in server mode on the used notebook. The iperf tool has been run on the Ubuntu 11.04 Linux distribution. Further, we used the latest available version 2.03 of the iperf tool.

iPerf for Android

The Android Market contains a porting of the iperf application for Android[2]. The application consists of a precompiled iperf binary for ARM and a minimalistic user interface. For our measurement, we used version 2.03 of *iPerf for Android*, which we downloaded from the official Android Market.

7.2 Scenarios

We created several test scenarios that simulate different configurations of our user-controlled Internet connections. These scenarios simulate several installed applications with restricted Internet access and lead to several rules in the Linux firewall. In our experiment, we simulated several extreme cases, for example, that one application contains a large amount of rules or that a large amount of applications with restricted Internet access is installed on the device. These extreme cases allow us to estimate the performance decrease caused by our implementation.

The general firewall design of all scenarios is applicable in Figure 7.1. The design is similar to the firewall design that we presented in Section 6.3.2. However, the rules are

[1] Official website available at `http://iperf.sourceforge.net/`.

[2] Available at `https://market.android.com/details?id=com.magicandroidapps.iperf`.

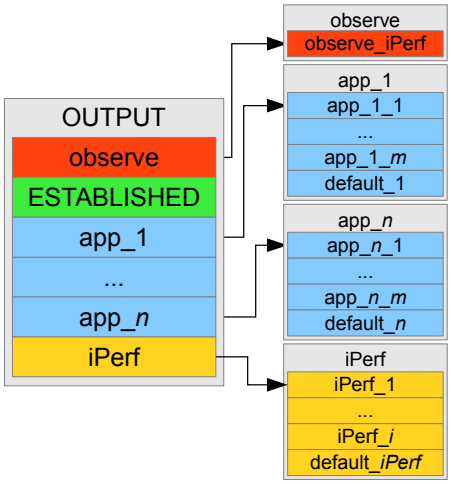


Figure 7.1: Firewall design for our test scenarios.

arranged in a way that allows us to measure the performance in a controlled manner. The design simulates the configuration of n applications plus the iPerf application. Thereby, each of the application chains contains m rules plus a default rule. The chain of the iPerf application contains i rules plus its default rule. Our experiments also include measurements for the observation mode of our solution. To inspect all network packets of the iPerf application, a corresponding rule is added to the `observe` chain. We also test the impact of the *established rule* that automatically accepts all network packets that belong to a connection that has already been accepted by the firewall. To measure the performance, we add the ESTABLISHED rule before the first application rule into the OUTPUT chain.

Currently, our Ucicia implementation does not allow to automatically create a large amount of rules and is only configurable through the user interface. Therefore, we did not configure the firewall through our tool. Instead, we configured the Linux firewall manually via a separate bash script. However, the design of the rules is identically equal to the rules that can be created through the UI.

We compare the network throughput of five different scenarios:

1. *Baseline*,
2. *several applications*,
3. *several rules*,
4. *dynamic Internet connection*, and
5. *observation mode*.

For (2), (3), and (4), we tested the performance with and without the established rule. We refer to these scenarios *without* the established rule as (2a), (3a), and (4a) and to those *with* the established rule as (2b), (3b), and (4b). For (5) we conducted three

different cases (5a), (5b), and (5c), to narrow down the performance bottleneck that we observed in our first measurement. In total, we examined ten different test cases.

7.2.1 Baseline

To simulate the behavior of the stock Android firmware, we emptied the Linux firewall completely. Without any firewall rules, our firmware behaves like the stock Android firmware and, therefore, we should not have any impact on the performance of the network connection. We use this scenario as the baseline of our experiment, such that we treat the network throughput of this scenario as the highest possible value and use it to compare it against other measurements.

For the sake of completeness, the iptables configuration of this scenario (and therefore of the empty firewall) is available in Appendix B.1. We extracted this report via the `iptables -S` command that prints out the current firewall configuration.

7.2.2 Several applications

Our first test scenario tested the performance when several applications with restricted Internet access are installed on the device. We simulated 200 applications with each having 21 filtering rules in its chain, but without any further rules for the iPerf application (i. e. $n = 200$, $m = 20$, and $i = 0$). The observation mode was disabled, such that the observe chain of the firewall was empty.

The UID of the applications ranged from 100 to 299. We created 20 accepting port rules for each application and set the default rule of the chain to reject all packets. The IP addresses of the accepting rules ranged from 209.85.148.10 to 209.85.148.19 and we used every address twice, once for TCP and once for UDP. The destination port was in each of the twenty cases set to port 80, the default port for HTTP connections.[3]

The chain for the iPerf application did not contain any further rules except for the default rule. The default rule was an accepting rule, such that the application is able to send out packets to the iperf server. We tested this scenario twice, first without the established rule and second with the established rule to compare the impact of this rule. The chain design is available in Appendix B.2.

7.2.3 Several rules

In the second test scenario, we tested the performance of the firewall in case each packet has to be compared against many IP based rules. We created 400 rules for the iPerf application and did not add rules for any other application (i. e. $n = m = 0$ and $i = 400$). We again left the observe chain untouched and tested the scenario twice with and without the established rule.

[3] We note here that the number of rules for the applications should not have any impact on the performance, since they should not be checked by the firewall anyway. Only the 200 rules in the OUTPUT chain are checked for an outgoing packet of the iPerf application. However, we wanted to create a realistic scenario and therefore also created those rules.

The IP addresses of the rules ranged from 209.85.148.10 to 209.85.148.209 and we again created one TCP and one UDP rule for each IP address. For each of the rules, the destination port was set to port 80 and the target of the rules was set to reject the connection.

Consequently, for each outgoing packet of the iPerf application, the Linux firewall has to compare it against 400 rules before the packet is finally accepted by the default rule of the iPerf application. The chain design is available in Appendix B.3.

7.2.4 Dynamic Internet connection

The third test scenario tested the performance of the dynamic Internet mode.Therefore, we queued all network packets of the iPerf application and handled them in the ucicia_queue user space application of our implementation. Normally, the user has to manually approve this connection via the UI. In order to test the performance automatically, we modified the framework service and automatically accept all incoming requests. This approach simulates the user interaction and ensures that we do not measure the delay that is introduced by the interaction with the UI.

In this scenario, we do neither have any application rules nor any additional rules for the iPerf application (i. e. $n = m = i = 0$). The only important rule is the default rule of the iPerf application that queues all packets and lets them handle in user space. We again did not have any observation rule and we tested the scenario with and without the established rule. The chain design is available in Appendix B.4.

7.2.5 Observation mode

The last test scenario tested the observation mode of our implementation. We added an observation rule for the iPerf application, set the default rule of the iPerf application to accepting, and enabled the established rule. There are no further rules defined, neither for additional applications nor for the iPerf application (i. e. $n = m = i = 0$). The chain design is available in Appendix B.5.

7.3 Evaluation

For each of the ten test cases, we conducted three measurements to measure the network throughput. We measured the number of transmitted data for 60 seconds. This time window should be large enough to compensate any variation in the wireless communication. We started the server on the notebook without any further configuration by running:

```
iperf -s
```

On the client side, we established a TCP connection to the server for 60 seconds and let the client output intermediate results each five seconds. Consequently, we started the iPerf client with the following arguments:

```
iperf -c 192.168.178.20 -i 5 -t 60
```

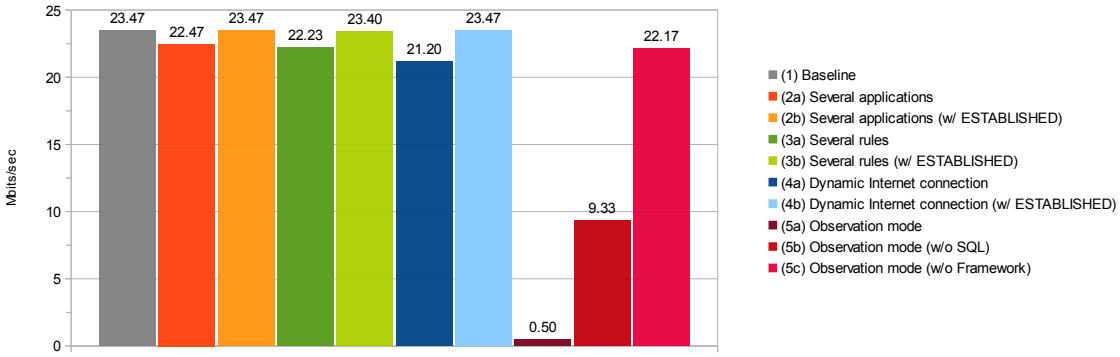


Figure 7.2: Measured average network throughput for each of the ten test cases.

We measured the performance of each test case three times and averaged over these three values. However, the dispersion between the values was not very high, except for the test case (5a) in which we measured 548, 468, and 479 kbit/s.

7.3.1 Interpretation

The results of our experiment are depicted in Figure 7.2. For the baseline of our experiment, we measured an average network throughput of 23.47 Mbit/s. We can see that, except for the observation mode, the average throughput stayed above 21 Mbit/s for all our measurements.

One important fact, we observed during our measurements is, that the decrease of the average throughput is negligible if we enable the established rule. Consequently, communications that transfer a large amount of data (e. g. file downloads) are not slowed down by our firewall approach.

If we do not enable the established rule, the average throughput decreases by about 1 Mbit/s for the scenario (2a) and (3a), and by about 2.2 Mbit/s for the dynamic Internet connection (4a). These test cases without the established rule represent the situation where several small connections are established and all network packets pass through the whole firewall chain. We can see that even for the dynamic Internet connection (in which case all packets are handled in user space), the average throughput does not decrease dramatically. We only loose about 5 to 10 percent of the network throughput.

However, the decrease for the observation mode was tremendous and lead to an average throughput of 0.5 Mbit/s. We tried to locate the cause for this large decrease and conducted a second measurement (5b). In this experiment, we disabled the connection to the database in order to check whether the database access slows down the communication. Without the database access, we measured an average throughput of 9.33 Mbit/s. In a last measurement (5c), we further disabled the communication to the Android framework, such that packets are only handled in the native application. This lead to an average throughput of 22.17 Mbit/s, which is almost the maximal achievable throughput.

This leads us to the assumption that handling of network packets in the user space

does not lead to drastic performance problems. However, the communication with the Android framework should be kept at a minimum. Both, dynamic Internet connections and the observation mode communicate with the framework, but we only see a drastic decrease for the observation mode. This difference is due to the caching of the user's decision for the dynamic Internet connection. The caching approach allows us to reduce the communication with the framework. Consequently, the dynamic Internet connection achieves a much better performance.

We conclude that our implementation of user-controlled Internet connections does not raise any performance issues that affect the every-day use of the mobile device. Our experiments show that a drastic decrease of the performance is limited to the observation mode. However, we assume that the performance could be raised to an acceptable level by accessing the database directly within the native application in order to avoid the current framework communication.

8 Related work

The open source character of Android make it an interesting target for current mobile security research. The source code of almost[1] all security relevant features of Android is available at the Android Open Source Project. This availability of the source code allows to modify the operating system and to test the modifications on one of the available developer phones. Further, the documentation of Android contains detailed information about the system design and about the applied security model.

In addition to the official documentation, there exist several works [24, 15, 45] that address the security model of Android in depth from the viewpoint of a developer. Further research related to mobile security on Android can be categorized into works that concentrates on *detecting malicious behavior* and into works that concentrates on *enhancing Android's security model*.

Malicious Behavior Detection

Enck et al. [22] implemented a Dalvik decompiler that allows to recover the Java source code of an Android application. Using the recovered source code, they analyzed 1100 applications from the Android Market and inspected them for malicious behavior. Decompiling the source code allowed them to use existing code analysis tools and to inspect some parts of the application manually. Enck et al. did not find any evidence for malicious behavior in the inspected applications. However, privacy sensitive information (e. g. the IMEI that uniquely identifies the phone) are misused by many applications.

The official Android API allows third-party developer to interact with the underlaying hardware. The access to these interfaces is in general protected by Android's permission

[1] Some of the Google related applications (e. g. the Android Market application) are not open sourced and therefore, can not be inspected in detail.

system. Felt et al. [25] analyzed which permissions are required to access the different interfaces of the API; whether the official API documentation reflects these requirements; and whether developers follow the principle of least privilege. By using automated testing techniques they identified 1207 methods in the API that are protected by a permission check and created a permission map that assigns each API call the required permissions. In contrast to their findings, only 78 methods of the API are documented as requiring a permission. Felt et al. built a static analysis tool *Stowaway* that identifies each call to the Android API. By using the generated permission map, Stowaway then extracts the minimum number of required permissions for an application and compares it to the requested permissions of the application. In their study, Felt et al. applied Stowaway to 940 applications of the Android Market and identified 35.8 % of the applications to be overprivileged. But 95 % of these applications requested only four or less unnecessary permissions, so that Felt et al. conclude that developers try to follow the principle of least privilege, but are often not able to do so.

Grace et al. [34] analyzed the firmware of eight Android phones from different manufacturers. They used static analysis techniques to reveal capability leaks that allow to circumvent the permission model of Android. These capability leaks allow applications to perform potential dangerous actions without holding the appropriate permission by exploiting flaws in the preinstalled applications. The results of Grace et al. show that phones with more preinstalled applications tend to contain more capability leaks and consequently leak up to eight of the stock permissions.

SCanDroid [29] uses data-flow analysis to identify potentially malicious applications by analyzing the source code of applications. It can track the flow of privacy sensitive information and detects if information leaves the application in an unintended manner (e. g. over the network). Similarly, *TaintDroid* [21] tracks the information-flow of Android applications in realtime on the device. This realtime tracking allows to inspect applications even if the source code is not available.

Enck et al. [23] analyzed different security threats on mobile devices with respect to the permissions that are required in order to execute the malicious behavior. From their analysis, they extracted several rules that identify dangerous combinations of permission requests. Instead of analyzing the information-flow of an application, their tool (*Kirin*) denies the installation of potentially malicious applications by comparing the requested permissions against the extracted rules. As an example, one of these rules states that an application is not allowed to request the GPS permission (`ACCESS_FINE_LOCATION`) in combination with the `INTERNET` permission and the permission to automatically start at boot time (`RECEIVE_BOOT_COMPLETE`). This rule protects the user from location tracking applications that send the current location to a remote server. Whenever a new application gets installed the requested permissions of the application are extracted from its `AndroidManifest.xml`. Then, Kirin compares these permissions with the global security rules and cancels the installation in case one of the rules is violated.

Schlegel et al. [48] introduce *Soundcomber*, a stealthy and context-aware Trojan for Android. Soundcomber requests the permission to access the microphone of the device, but simultaneously avoids requesting the Internet permission. Since requesting only one permission is not assumed to be dangerous, the protection techniques of Kirin can

not defend against this type of Trojan. However, the Trojan listens to specific phone calls and tries to extract credit card information directly on the phone. Afterwards, Soundcomber utilizes other third-party applications (e. g. the browser) to transmit the extracted information to a remote server. Further, Schlegel et al. present several covert channels that can be used to stealthy transmit the data to an allied application that requested the required Internet permission.

Davi et al. [18] present *privilege escalation attacks* on Android that allow a non-privileged application to access a resource by utilizing a privileged application. They showed that an application can send SMS messages without requesting any permissions by exploiting a vulnerability inside another application. Bugiel et al. [14] introduce *XManDroid* to mitigate against these privilege escalation attacks by analyzing the transitive permission usage in realtime. Their approach also defends against the covert channels used by the Soundcomber Trojan.

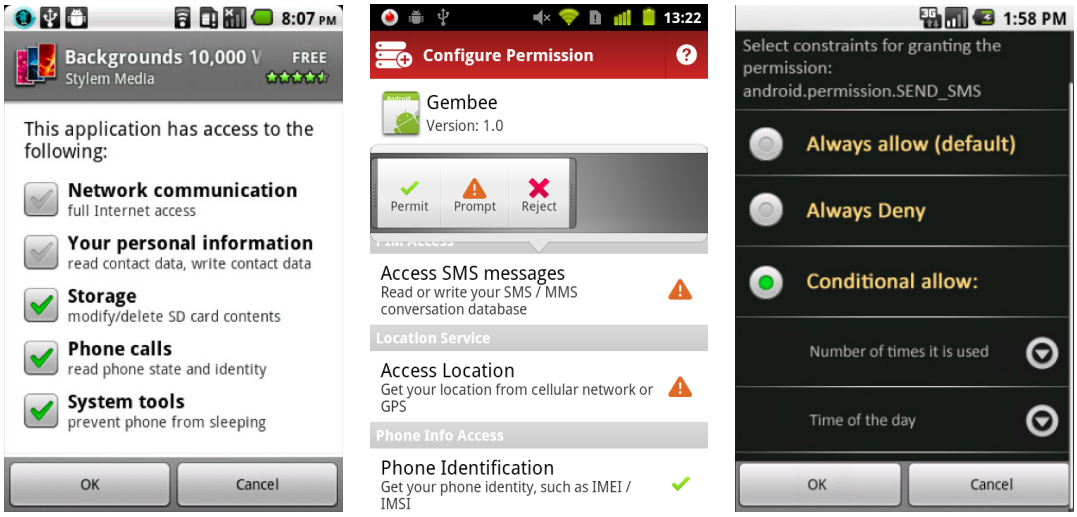
Enhancement of Android's security model

Barrera et al. [11] analyzed the top 50 free applications from each of the 22 categories in the Android Market. They used a specific algorithm to visualize the permission requests of the 1100 sample applications. Their study revealed different characteristics about used permissions in Android. Several combinations of permissions are often requested in common (e. g. *read_contacts* and *write_contacts*). Further, the majority of the tested applications requested the Internet permission which lead to the conclusion that the Internet permission of Android does not provide sufficient fine-grained access to the Internet.

Mueller and Butler [41] proposed a more flexible implementation of Android's permission model, named *Flex-P*. By default users have only two possibilities to influence the permissions that are granted to an application: They either install the application and give it *all* requested permissions or they decide to not install the application at all. The modified permission model allows users to revoke requested permissions at the time of installation, as shown in Figure 8.1a. Additionally, users are able to change the granted permissions at any time to either give the application more permissions or the revoke already granted permissions. But since the permission model assumes that all requested permissions get granted at install-time, an application with less permissions than requested may crash.

There also exists several applications in the Android market that allow to revoke permissions. The revocation of the permission happens *after* an application has been installed, because hooking into the installation process requires modifications in the firmware. One of these applications is the *LBE Privacy Guard* [2]. In addition to accepting and revoking permission of an application, the LBE Privacy Guard allows to postpone the decision to the point where the permission is actually required. Every time the application requires a specific permission the user gets informed and needs to grant the permission dynamically. Figure 8.1b shows that this works for a subset of the default

[2] https://market.android.com/details?id=com.lbe.security



(a) Flex-P (b) LBE Privacy Guard (c) Apex

Figure 8.1: Different approaches to extend the Android permission system.

permissions; for example, the permission to get the current location via GPS or the permission to read and write to the SMS conversation database.

Apex [42] extends the Android permission model with user-defined runtime constraints. Therefore, Nauman et al. modified the default permission check in the Android operating system and added a framework to evaluate context aware constraints. Figure 8.1c shows the extended permission assignment. As an example they implemented two simple constraints. The first one allows to restrict the number of times a resource is used per day, which is, for example, useful to limit the number of send SMS. The other one allows to restrict the use of a permission to a specific time window of the day. For example, accessing the GPS sensor is not allowed between 10 p.m. and 7 a.m. while the user is sleeping. As long as an application does not violate these constraints it works as desired, but it may crash if any of the constraints is violated. These fine-grained constraints also have a rather large impact on the execution time of a permission check. As a consequence the check whether an application is allowed to send an SMS increases from 34 ms to 103 ms.

A even more contextual extension of the Android permission model is *CRePE* [17]. CRePE introduces a context-related security policy that grants and revokes certain permissions based on the context of the device. This, for example, allows to enable the Bluetooth interface when the user is at home, but automatically disable it as soon as the device is not within a defined environment. In addition, CRePE respects rules defined by trusted third parties, so that, for example, the camera functionality is automatically disabled inside a museum.

Another approach by Zhou et al. [56], named *TISSA*, concentrates on protecting predefined personal information on the phone; this includes the current location, the phone identity (IMEI), the contact list, or the call log. All these personal information is by

default protected by several different permissions. But as seen before, revoking any of these permissions may lead to an unstable version of an application. TISSA allows to restrict the access to the personal information within four different levels: *Trusted* allows the application to access the real data as before; *Anonymized* returns an anonymized version (e. g. a random location within a predefined radius of some kilometers); *Bogus* returns some fake and maybe random data (e. g. a randomly generated identification number of the phone); and *Empty* pretends that the requested information is not available or empty (e. g. an empty contacts list). Since no exception is thrown because of missing permissions and since the returned information looks totally valid, a crash of an application—not allowed to access the real privacy sensitive information—is rather unlikely. But TISSA can only protect predefined personal information. If an application requests to enter some login credentials or other data directly, there is no chance for TISSA to protect this information.

MockDroid [12] and AppFence [36] pursue a similar goal and provide shadow data instead of the actual information. In addition to providing fake data, AppFence includes and extends the functionality of TaintDroid and blocks network connections containing private data. To test the impact of AppFence, Hornyack et al. used a semi-automatic approach to compare the behavior of 110 popular permission-hungry applications running on the stock Android firmware with the behavior on their modified firmware. They found that their modifications do not cause side effects in 66 % of the tested applications. This results show that the majority of the applications can be executed with shadowed private information.

The mentioned extensions of the permission model protect the system from malicious applications. In contrast, Ongtang et al. [44] introduce *Saint*, an application-centric extension of Android's security model. Saint allows an application to define additional constraints that define whether access from other applications is allowed or not. These runtime constraints can help to mitigate against the privilege escalation attacks that we mentioned above. However, application developers has to take care to define the appropriate constraints, which, for example, ensure that location information is only leaked to applications that requested the ACCESS_LOCATION permission themselves.

Another paper by Fragkaki et al. [26] has just been released in late November. Their goal is to build an abstract model of Android's permission system. Analyzing this model allows to reveal design and implementation flaws in the existing system. Thereupon, Fragkaki et al. propose an extended permission model that mitigates against privilege escalation attacks, undesired information flow, and allows fine-grained runtime delegation of permissions.

9 Conclusion

In this work, we refined Android's permission model in order to support fine-grained Internet permission requests. The refined permission model allows application developers to request the Internet permission based on the destination of outgoing connections. We further give the user more control over the requested Internet permissions and allow him to reject the Internet permission without canceling the installation. This makes it possible to install applications in a sandbox that does not allow Internet connections. In addition, we make the Internet permission more dynamic, such that the user can postpone the decision about the permission request until a connection with the Internet is actually established.

To demonstrate the refined permission model in practice, we modified the stock Android firmware and implemented user-controlled Internet permissions in Android. By using the Linux firewall, we enforce the refined permission model and therefore allow the user to grant fine-grained Internet permissions. We tested our implementation on a Nexus One developer phone and a performance analysis of our implementation showed that the decrease in the average network throughput is low.

Limitations The underlying network infrastructure causes some limitations regarding the fine-grained Internet permission. On the one hand, all network communication is based on IP addresses in order to identify the source and destination of a connection. On the other hand, domain names are in general used to look up the IP address of some service via a human-readable name. Especially for web hosting services several domain names point to the same IP address and different persons provide their service on the same server. Consequently, by granting an application access to such an IP address that hosts several services, the application is able to access all of them and there is no way to further distinguish those services on the network layer. However, we assume that

security critical services are hosted on a dedicated server that is only controlled by the provider of the service.

Future work Regarding the dynamic Internet permission, a future goal would be to improve the expressiveness of the dynamic Internet request by determining the domain name of outgoing requests in a reliable manner. The domain name of a connection helps to better identify the destination of the connection, because everybody using the Internet is more used to enter domain names instead of IP addresses. The domain name of an IP address could be determined by some sort of reverse DNS lookup that either uses the existing reverse lookup techniques of DNS or provides a similar lookup table directly on the mobile phone.

Another future goal would be to expand the current fine-grained Internet permission model to other built-in permissions. Especially the `CALL_PHONE` and the `SEND_SMS` permission would profit from a further refinement. Fine-grained permission requests based on the phone number would give the user more insight into the purpose of the permission request and they would make it obvious whether an application tries to contact costly premium services or not.

Bibliography

[1] ABI Research. *Android Overtakes Apple with 44% Worldwide Share of Mobile App Downloads*. Oct. 24, 2011. URL: http://www.abiresearch.com/press/3799-Android+Overtakes+Apple+with+44%25+Worldwide+Share+of+Mobile+App+Downloads (visited on 10/28/2011).

[2] Android Open Source Project. *Android Security Overview*. URL: http://source.android.com/tech/security/index.html (visited on 11/18/2011).

[3] Android Open Source Project. *Notes on the implementation of encryption in Android 3.0*. URL: http://source.android.com/tech/encryption/android_crypto_implementation.html (visited on 11/19/2011).

[4] Android Project. *Android 3.0 Platform Highlights*. URL: http://developer.android.com/sdk/android-3.0-highlights.html (visited on 12/13/2011).

[5] Android Project. *Android Interface Definition Language (AIDL)*. URL: http://developer.android.com/guide/developing/tools/aidl.html (visited on 10/18/2011).

[6] Android Project. *Platform Versions: Current Distribution*. URL: http://developer.android.com/resources/dashboard/platform-versions.html (visited on 11/06/2011).

[7] Android Project. *What is Android?* URL: http://developer.android.com/guide/basics/what-is-android.html (visited on 10/18/2011).

[8] Apple Inc. *App Store Review Guidlines*. URL: http://developer.apple.com/appstore/guidelines.html (visited on 07/25/2011).

[9] Michael Backes, Sebastian Gerling, and Philipp von Styp-Rekowsky. "A Local Cross-Site Scripting Attack against Android Phones". 2011.

[10] Hugo Barra. *Android: momentum, mobile and more at Google I/O*. May 10, 2011. URL: http://googleblog.blogspot.com/2011/05/android-momentum-mobile-and-more-at.html (visited on 07/25/2011).

[11] David Barrera, Hilmi Günes Kayacik, Paul C. van Oorschot, et al. "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*. CCS '10 (Chicago, IL, USA, Oct. 4–8, 2010). Ed. by Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov. ACM, 2010, pp. 73–84.

[12] Alastair R. Beresford, Andrew Rice, Nicholas Skehin, et al. "MockDroid: trading privacy for application functionality on smartphones". In: *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*. HotMobile '11 (Phoenix, AZ, USA, Mar. 1–2, 2011). To appear. ACM, 2011.

[13] Dan Bornstein. *2008 Google I/O Session Videos and Slides. Dalvik VM Internals*. URL: `https://sites.google.com/site/io/dalvik-vm-internals` (visited on 10/18/2011).

[14] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, et al. *XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks*. Tech. rep. TR-2011-04. Technische Universität Darmstadt, 2011.

[15] Jesse Burns. *Developing Secure Mobile Applications for Android. An Introduction to Making Secure Android Applications*. Tech. rep. iSec Partners, Oct. 2008.

[16] Rich Cannings. *An Update on Android Market Security*. Mar. 5, 2011. URL: `http://googlemobile.blogspot.com/2011/03/update-on-android-market-security.html` (visited on 11/04/2011).

[17] Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. "CRePE: Context-Related Policy Enforcement for Android". In: *Information Security. 13th International Conference, Revised Selected Papers*. ISC 2010 (Boca Raton, FL, USA, Oct. 25–28, 2010). Ed. by Mike Burmester, Gene Tsudik, Spyros S. Magliveras, et al. Vol. 6531. Lecture Notes in Computer Science. Springer, 2011, pp. 331–345.

[18] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, et al. "Privilege Escalation Attacks on Android". In: *Information Security. 13th International Conference, Revised Selected Papers*. ISC 2010 (Boca Raton, FL, USA, Oct. 25–28, 2010). Ed. by Mike Burmester, Gene Tsudik, Spyros S. Magliveras, et al. Vol. 6531. Lecture Notes in Computer Science. Springer, 2011, pp. 346–360.

[19] Michael DeGusta. *Android Orphans: Visualizing a Sad History of Support*. Oct. 26, 2011. URL: `http://theunderstatement.com/post/11982112928/android-orphans-visualizing-a-sad-history-of-support` (visited on 11/06/2011).

[20] Scott Delap. *Google's Android SDK Bypasses Java ME in Favor of Java Lite and Apache Harmony*. Nov. 12, 2007. URL: `http://www.infoq.com/news/2007/11/android-java` (visited on 10/18/2011).

[21] William Enck, Peter Gilbert, Byung-gon Chun, et al. "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones". In: *9th USENIX Symposium on Operating Systems Design and Implementation*. OSDI '10 (Vancouver, BC, Canada, Oct. 4–16, 2010). Ed. by Remzi H. Arpaci-Dusseau and Brad Chen. USENIX Association, 2010, pp. 393–407.

[22] William Enck, Damien Octeau, Patrick McDaniel, et al. "A study of Android application security". In: *Proceedings of the 20th USENIX Security Symposium*. USENIX Security '11 (San Francisco, CA, USA, Aug. 8–12, 2011). 2011.

[23] William Enck, Machigar Ongtang, and Patrick Drew McDaniel. "On lightweight mobile phone application certification". In: *Proceedings of the 16th ACM Conference on Computer and Communications security*. CCS '09 (Chicago, IL, USA, Nov. 9–13, 2009). Ed. by Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis. ACM, 2009, pp. 235–245.

[24] William Enck, Machigar Ongtang, and Patrick Drew McDaniel. "Understanding Android Security". In: *IEEE Security & Privacy* 7.1 (2009), pp. 50–57.

[25] Adrienne Porter Felt, Erika Chin, Steve Hanna, et al. "Android permissions demystified". In: *Proceedings of the 18th ACM Conference on Computer and Communications security*. CCS '11 (Chicago, IL, USA, Oct. 17–21, 2011). Ed. by Yan Chen, George Danezis, and Vitaly Shmatikov. ACM, 2011, pp. 627–638.

[26] Elli Fragkaki, Lujo Bauer, and Limin Jia. *Modeling and Enhancing Android's Permission System*. Tech. rep. CMU-CyLab-11-020. CyLab, Carnegie Mellon University, 2011.

[27] Clemens Fruhwirth. *New methods in hard disk encryption*. 2005.

[28] F-Secure. *Warning On Possible Android Mobile Trojans*. Jan. 11, 2010. URL: `http://www.f-secure.com/weblog/archives/00001852.html` (visited on 12/05/2011).

[29] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. *SCanDroid: Automated Security Certification of Android Applications*. Tech. rep. CS-TR-4991. Department of Computer Science, University of Maryland, College Park, 2009.

[30] Gartner Inc. *Gartner Says Android to Become No. 2 Worldwide Mobile Operating System in 2010 and Challenge Symbian for No. 1 Position by 2014*. Sept. 10, 2010. URL: `http://www.gartner.com/it/page.jsp?id=1434613` (visited on 10/18/2011).

[31] Gartner Inc. *Gartner Says Android to Command Nearly Half of Worldwide Smartphone Operating System Market by Year-End 2012*. Apr. 7, 2011. URL: `http://www.gartner.com/it/page.jsp?id=1622614` (visited on 10/18/2011).

[32] Gartner Inc. *Gartner Says Sales of Mobile Devices Grew 5.6 Percent in Third Quarter of 2011; Smartphone Sales Increased 42 Percent*. Nov. 15, 2011. URL: `http://www.gartner.com/it/page.jsp?id=1848514` (visited on 12/12/2011).

[33] Google Inc. *Android Market: Publish*. URL: `http://market.android.com/publish/` (visited on 07/25/2011).

[34] Michael Grace, Yajin Zhou, Zhi Wang, et al. *Detecting Capability Leaks in Android-based Smartphones*. Tech. rep. TR-2011-15. North Carolina State University, 2011.

[35] Dianne Hackborn. *OpenBinder*. URL: `http://www.angryredplanet.com/~hackbod/openbinder` (visited on 10/18/2011).

[36] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, et al. ""These Aren't the Droids You're Looking For": Retrofitting Android to Protect Data from Imperious Applications". In: *Proceedings of the 18th ACM Conference on Computer and Communications security*. CCS '11 (Chicago, IL, USA, Oct. 17–21, 2011). Ed. by Yan Chen, George Danezis, and Vitaly Shmatikov. ACM, 2011.

[37] Kaspersky Lab. *ZeuS-in-the-Mobile for Android*. June 12, 2011. URL: `http://www.securelist.com/en/blog/208193029/ZeuS_in_the_Mobile_for_Android` (visited on 12/13/2011).

[38] James F. Kurose and Keith W. Ross. *Computer networking : a top-down approach*. 5th ed. Pearson Education, 2010.

[39] Lookout. *Security Alert: Fake Netflix App Aids Phishing*. Oct. 13, 2011. URL: `http://blog.mylookout.com/2011/10/security-alert-fake-netflix-app-aids-phishing/` (visited on 12/06/2011).

[40] McAfee Labs. *McAfee Threats Report: Second Quarter 2011*. Aug. 23, 2011. URL: `http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q2-2011.pdf` (visited on 11/04/2011).

[41] Kurt Mueller and Kevin Butler. *Poster: Flex-P: Flexible Android Permissions*. 2011.

[42] Mohammad Nauman, Sohail Khan, and Xinwen 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*. ASIACCS '10 (Beijing, China, Apr. 13–16, 2010). Ed. by Dengguo Feng, David A. Basin, and Peng Liu. ACM, 2010, pp. 328–332.

[43] Network Working Group. *RFC 1034: DOMAIN NAMES - CONCEPTS AND FACILITIES*. 1987. URL: `https://tools.ietf.org/html/rfc1034` (visited on 12/06/2011).

[44] Machigar Ongtang, Stephen McLaughlin, William Enck, et al. "Semantically Rich Application-Centric Security in Android". In: *Proceedings of the 2009 Annual Computer Security Applications Conference*. ACSAC '09 (Honolulu, HI, USA, Dec. 7–11, 2009). IEEE Computer Society, 2009, pp. 340–349.

[45] C. Enrique Ortiz. *Understanding security on Android. Enhance application security with sandboxes, application signing, and permissions*. 2010. URL: `http://www.ibm.com/developerworks/opensource/library/x-androidsecurity/index.html` (visited on 12/18/2011).

[46] Kyle Randolph. *Inside Adobe Reader Protected Mode – Part 1 – Design*. Adobe Systems Inc. Oct. 5, 2010. URL: `http://blogs.adobe.com/asset/2010/10/inside-adobe-reader-protected-mode-part-1-design.html` (visited on 11/03/2011).

[47] William Sallings and Lawrie Brown. *Computer Security: Principles and Practice*. 1st ed. Pearson, 2008.

[48] Roman Schlegel, Kehuan Zhang, Xiao-yong Zhou, et al. “Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones”. In: *Proceedings of the Network and Distributed System Security Symposium*. NDSS 2011 (San Diego, CA, USA, Feb. 6–9, 2011). The Internet Society, 2011.

[49] Bruce Schneier. *Secrets and Lies: Digital Security in a Networked World*. 1st ed. Wiley, 2004.

[50] Justin Shapcott. *Updates, or lack thereof, on the Android Update Alliance*. Android and Me. Aug. 30, 2011. URL: `http://androidandme.com/2011/08/news/updates-or-lack-thereof-on-the-android-update-alliance/` (visited on 11/06/2011).

[51] R. Spenneberg. *Linux-Firewalls mit Iptables & Co*. Die Linux-Security-Box. Addison-Wesley, 2006.

[52] Symantec. *Will Your Next TV Manual Ask You to Run a Scan Instead of Adjusting the Antenna?* Oct. 13, 2011. URL: `http://www.symantec.com/connect/blogs/will-your-next-tv-manual-ask-you-run-scan-instead-adjusting-antenna` (visited on 12/06/2011).

[53] Technology Review. *How Android Security Stacks Up*. Apr. 1, 2010. URL: `http://www.technologyreview.com/printer_friendly_article.aspx?id=24944` (visited on 11/03/2011).

[54] The Chromium Projects. *Sandbox*. URL: `http://dev.chromium.org/developers/design-documents/sandbox` (visited on 11/03/2011).

[55] Alberto Vildosola. *Google partners with manufacturers and carriers to speed up Android updates*. Android and Me. May 10, 2011. URL: `http://androidandme.com/2011/05/uncategorized/google-partners-with-manufacturers-and-carriers-to-speed-up-android-updates/` (visited on 11/06/2011).

[56] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, et al. “Taming Information-Stealing Smartphone Applications (on Android)”. In: *Trust and Trustworthy Computing. 4th International Conference on Trust and Trustworthy Computing*. TRUST 2011 (Pittsburgh, PA, USA, June 22–24, 2011). Ed. by Jonathan M. McCune, Boris Balacheff, Adrian Perrig, et al. Vol. 6740. Lecture Notes in Computer Science. Springer, 2011, pp. 93–107.

Appendices

A
Iptables commands

```
# Creation of observation chain
iptables -N ucicia_app_observe
iptables -I OUTPUT 1 -m mark --mark 0 -j ucicia_app_observe
# Creation and removal of observation rules for an application
iptables -A ucicia_app_observe -m owner --uid-owner <uid> -j NFQUEUE
    --queue-num 2
iptables -D ucicia_app_observe -m owner --uid-owner <uid> -j NFQUEUE
    --queue-num 2

# Creation of the ESTABLISHED rule
iptables -I OUTPUT 1 -m state --state ESTABLISHED -j ACCEPT

# Creation and removal of new application chain
iptables -N ucicia_app_<uid>
iptables -X ucicia_app_<uid>
iptables -F ucicia_app_<uid>

# Creation and removal of corresponding rule
# for application chain in OUTPUT chain
iptables -A OUTPUT -m owner --uid-owner <uid> -j ucicia_app_<uid>
iptables -D OUTPUT -m owner --uid-owner <uid> -j ucicia_app_<uid>

# Filtering rules of an application
iptables -A ucicia_app_<uid> -j <target> [--queue-num 1] [-d
    <destination>] [-p <tcp/udp> --destination-port <port>]
# Default rule of an application
iptables -A ucicia_app_<uid> -j <target> [--queue-num 1]
```

B
Iptables configurations

B.1 Baseline

```
# Behavior of built-in chains
-P INPUT ACCEPT
-P FORWARD ACCEPT
-P OUTPUT ACCEPT
```

B.2 Several applications

```
# Behavior of built-in chains
-P INPUT ACCEPT
-P FORWARD ACCEPT
-P OUTPUT ACCEPT

# Creation of custom chains
-N ucicia_app_10049
-N ucicia_app_100
-N ucicia_app_101
# ...
-N ucicia_app_299
-N ucicia_app_observe

# Initialization of OUTPUT chain
-A OUTPUT -m mark --mark 0x0 -j ucicia_app_observe
# Next line only present when testing with ESTABLISHED rule
-A OUTPUT -m state --state ESTABLISHED -j ACCEPT
-A OUTPUT -m owner --uid-owner 299 -j ucicia_app_299
-A OUTPUT -m owner --uid-owner 298 -j ucicia_app_298
# ...
-A OUTPUT -m owner --uid-owner 101 -j ucicia_app_101
```

```
-A OUTPUT -m owner --uid-owner 100 -j ucicia_app_100
-A OUTPUT -m owner --uid-owner app_49 -j ucicia_app_10049

# Default rule of iPerf application
-A ucicia_app_10049 -j ACCEPT

# 20 ip rules + 1 default rule for application 100
-A ucicia_app_100 -d 209.85.148.10/32 -p tcp -m tcp --dport 80 -j ACCEPT
-A ucicia_app_100 -d 209.85.148.10/32 -p udp -m udp --dport 80 -j ACCEPT
-A ucicia_app_100 -d 209.85.148.11/32 -p tcp -m tcp --dport 80 -j ACCEPT
-A ucicia_app_100 -d 209.85.148.11/32 -p udp -m udp --dport 80 -j ACCEPT
# ...
-A ucicia_app_100 -d 209.85.148.19/32 -p tcp -m tcp --dport 80 -j ACCEPT
-A ucicia_app_100 -d 209.85.148.19/32 -p udp -m udp --dport 80 -j ACCEPT
-A ucicia_app_100 -j REJECT --reject-with icmp-port-unreachable

# ...

# 20 ip rules + 1 default rule for application 299
-A ucicia_app_299 -d 209.85.148.10/32 -p tcp -m tcp --dport 80 -j ACCEPT
-A ucicia_app_299 -d 209.85.148.10/32 -p udp -m udp --dport 80 -j ACCEPT
-A ucicia_app_299 -d 209.85.148.11/32 -p tcp -m tcp --dport 80 -j ACCEPT
-A ucicia_app_299 -d 209.85.148.11/32 -p udp -m udp --dport 80 -j ACCEPT
# ...
-A ucicia_app_299 -d 209.85.148.19/32 -p tcp -m tcp --dport 80 -j ACCEPT
-A ucicia_app_299 -d 209.85.148.19/32 -p udp -m udp --dport 80 -j ACCEPT
-A ucicia_app_299 -j REJECT --reject-with icmp-port-unreachable
```

B.3 Several rules

```
# Behavior of built-in chains
-P INPUT ACCEPT
-P FORWARD ACCEPT
-P OUTPUT ACCEPT

# Creation of custom chains
-N ucicia_app_10049
-N ucicia_app_observe

# Initialization of OUTPUT chain
-A OUTPUT -m mark --mark 0x0 -j ucicia_app_observe
# Next line only present when testing with ESTABLISHED rule
-A OUTPUT -m state --state ESTABLISHED -j ACCEPT
-A OUTPUT -m owner --uid-owner app_49 -j ucicia_app_10049

# 400 ip rules + 1 default rule for iPerf application
-A ucicia_app_10049 -d 209.85.148.10/32 -p tcp -m tcp --dport 80 -j
    REJECT --reject-with icmp-port-unreachable
-A ucicia_app_10049 -d 209.85.148.10/32 -p udp -m udp --dport 80 -j
    REJECT --reject-with icmp-port-unreachable
-A ucicia_app_10049 -d 209.85.148.11/32 -p tcp -m tcp --dport 80 -j
    REJECT --reject-with icmp-port-unreachable
-A ucicia_app_10049 -d 209.85.148.11/32 -p udp -m udp --dport 80 -j
```

```
    REJECT --reject-with icmp-port-unreachable
# ...
-A ucicia_app_10049 -d 209.85.148.209/32 -p tcp -m tcp --dport 80 -j
    REJECT --reject-with icmp-port-unreachable
-A ucicia_app_10049 -d 209.85.148.209/32 -p udp -m udp --dport 80 -j
    REJECT --reject-with icmp-port-unreachable
-A ucicia_app_10049 -j ACCEPT
```

B.4 Dynamic Internet connection

```
# Behavior of built-in chains
-P INPUT ACCEPT
-P FORWARD ACCEPT
-P OUTPUT ACCEPT

# Creation of custom chains
-N ucicia_app_10049
-N ucicia_app_observe

# Initialization of OUTPUT chain
-A OUTPUT -m mark --mark 0x0 -j ucicia_app_observe
# Next line only present when testing with ESTABLISHED rule
-A OUTPUT -m state --state ESTABLISHED -j ACCEPT
-A OUTPUT -m owner --uid-owner app_49 -j ucicia_app_10049

# Default rule of the iPerf application that queues all packets
-A ucicia_app_10049 -j NFQUEUE --queue-num 1
```

B.5 Observation mode

```
# Behavior of built-in chains
-P INPUT ACCEPT
-P FORWARD ACCEPT
-P OUTPUT ACCEPT

# Creation of custom chains
-N ucicia_app_10049
-N ucicia_app_observe

# Initialization of OUTPUT chain
-A OUTPUT -m mark --mark 0x0 -j ucicia_app_observe
-A OUTPUT -m state --state ESTABLISHED -j ACCEPT
-A OUTPUT -m owner --uid-owner app_49 -j ucicia_app_10049

# Observation mode rule for iPerf application
-A ucicia_app_observe -m owner --uid-owner app_49 -j NFQUEUE --queue-num
    2

# Default rule of the iPerf application that queues all packets
-A ucicia_app_10049 -j ACCEPT
```