Analyzing Covert Channels on Mobile Devices

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Analyzing Covert Channels on Mobile Devices

Master Thesis

Hubert Ritzdorf

Thursday 5th April, 2012

Advisors: Prof. Dr. S. Čapkun, C. Marforio

Department of Computer Science, ETH Zürich

Abstract

In this work we investigate the problem of stealthy communication between colluding applications on smartphones running the popular Android operating system. Through collusion, applications can cooperatively perform operations they would not be able to perform separately, thus escalating their privileges. This can result in privacy infringements and user data leakage. In order to collude, the two applications must communicate in a way that can bypass the application isolation put in place by the operating system.

Throughout this thesis we present different ways to bypass the isolation and thereby allow application collusion. As covert channels are by definition harder to implement but also harder to detect, we use them to create a circumvention that is harder to defeat. To understand the full extent of this problem, we implement very different overt and covert channels and analyse them by testing their throughput, bit-error percentage and synchronisation time. Using our implemented channels we analyse some previously existing countermeasures, in particular TaintDroid and XManDroid, and comment on the countermeasures strengths and limitations. Finally we use the lessons learned from implementing these channels and propose individual countermeasures, which can reduce the feasibility of creating such channels. In this scenario preventing hidden communication channels remains an open problem that we believe the research community should put more focus on.

Acknowledgments

I would like to take the opportunity to thank Prof. Srdjan Čapkun for introducing me to this interesting topic and creating a productive work environment that made my thesis an enjoyable and insightful experience.

I would also like to thank Claudio Marforio for his excellent supervision of my thesis and all the assistance and contributions he provided.

Futhermore I want to express gratitude towards Kevin Borgolte, Matthias Herrmann, Luka Malisa, Joel Reardon and Petar Tsankov for their helpful and enlightning thought exchange and last but not least for reviewing my thesis. The helpful effort of Der-Yeuan Yu and Nan Zhong during the final hours should also not go unnoticed.

Finally I want to thank Sven Bugiel and the members of System Security Lab of the Center for Advanced Security Research Darmstadt for the very interesting discussions, particularly about XManDroid.

Contents

Chapter 1

Introduction

Worldwide smartphones sales are rising rapidly and in the fourth quarter of 2011, Google Android [1] was running on more than half of the smartphones sold worldwide [17]. Additionally, a recent study states that almost every second Swiss owns a smartphone [9]. Therefore the security of these systems has become an important topic.

On Android smartphones, new applications can be easily installed through application markets and can therefore spread rapidly. At the time of this writing over 450,000 applications are available on Google Play [2]. The combination of the sheer number of applications mostly written by untrusted parties and the feature-rich environment of a smartphone, often holding corporate and private data, results in a lot of potential information leakage. A single application having access to sensitive data, such as SMS or Contacts, and access to the Internet, can leak sensitive data to any third party on the Internet. Indeed such leakage has been uncovered numerous times [12, 13]. As a consequence users have become aware of the potential dangers of applications with a powerful set of permissions. However it is unclear to which extent the application isolation can be circumvented so that two applications can act as one powerful, even though the user carefully checked their permissions. This can be achieved through collusion so that the two applications escalate their privileges as they combine their previously separated sets of permissions.

Previous work [24, 28] has identified this problem and provided different channels between colluding applications that can lead to privilege escalation. As a countermeasure, TaintDroid [12] presented a way of perceiving the leakage of sensitive information by tracking the information flow of such data within and between applications. However the focus has been explicitly limited to a certain group of channels. Additionally, XManDroid [4, 5] offered a system to detect communication between applications based on overt as well as some covert channels. It allows the policy-based filtering of inter-application communication to prevent certain undesirable data flows.

We provide an implementation and detailed analysis of previously described channels from Soundcomber [28] that use changes in volume or vibration settings to transmit information. Furthermore we describe, implement and analyse new channels. There are some simple channels that use the file system, Android communication mechanisms or standard socket communication. Additionally, we have more covert channels using meta data or side-effects of operations to communicate. Finally we also introduce a channel that makes use of an existing application. We study the practical differences when implementing these channels and do an analysis in terms of throughput, bit-error rate and synchronisation time. Using the analysis results, when tested on different phones with different APIs, we point towards the potential problems raised by covert channels. Using the information gathered during implementation and testing, we introduce possible countermeasures for channels, which have not been investigated so far.

Overall we analyse whether two less powerful and thereby seemingly harmless applications can pose a similar threat as a single, very powerful application. We therefore investigate the power and applicability of different channels as a mechanism to bypass the Android security model and allow applications to escalate their privileges by combining them. In contrast to most of the previous work, we mainly focus on covert channels as they are an often overlooked way to bypass standard security mechanisms. We additionally outline the strengths and weaknesses of the existing countermeasures, pointing towards possible improvements.

In Chapter 2 we present a brief overview of the Android operating system and the related work of this thesis. In Chapter 3 we describe the implemented channels, our testing environment and the performance of our channels. In Chapter 4 we explain the some previously created tools and evaluate their effectiveness against our channels. In Chapter 5 we propose and evaluate possible countermeasures we found against our channels and we conclude in Chapter 6.

Chapter 2

Background

In this chapter we present the reader with background information about the Android operating system and related work.

2.1 The Android Operating System

In this chapter we briefly outline the main aspects of the Android operating system while keeping focus on aspects that will be relevant within the context of this thesis.

The Android OS allows the installation of additional, third-party applications based on its powerful Application Programming Interface (API). Such applications can be programmed using Java, which is then executed inside the Dalvik VM. Additionally the applications can invoke native code using the Java Native Interface (JNI). As a smartphone is a very feature-rich device often containing a significant amount of sensitive data, the Android OS isolates the different applications from each other and from certain resources. Access to such resources has to be allowed by the user.

In order to allow the allocation of different *permissions* to different applications, Android offers a permission model that contains roughly 120 predefined permissions[1]. These permissions range from access rights to sensors, such as camera or microphone, over access rights to certain information, such as contacts, to admission of communication, such as web traffic or Bluetooth. One of the most important permissions is the `INTERNET` permission, which provides complete and unfiltered Internet access. Upon creation of the application, the author has to define the required permissions for his application. Upon installation the user can either choose to grant all the permissions and install the application or choose not to install the application at all.

[1] http://developer.android.com/reference/android/Manifest.permission.html

Once an application is installed, it is assigned a usually unique Unix user id and the permissions are permanently granted so that no further user interaction on this issue appears. The permissions are then enforced by both the kernel and the Dalvik VM so that native code as well as Java code is restricted according to the permissions. The kernel enforces some of the permissions by assigning the applications to certain groups and only allowing certain operations to members of the according group [22].

However, certain interaction between the applications is desirable from the principle of modularity and in order to simplify programming. Android provides a possibility for inter-process communication (IPC) called *intents*. The intents are objects that can be broadcasted on the system and allow the specification of an associated action upon reception. They allow to start up or notify other applications while providing additional data or can be used to trigger an action, such as opening a web site in a browser. As intents can trigger powerful actions, applications have the ability to restrict the set of applications allowed to send a special intent and can thereby protect their interfaces.

The smartphone usually comes with a certain amount of internal storage, which holds the initial applications and provides the default installation location for new applications. By default Android provides every application with a private directory on the internal storage. Additionally a lot of users prefer to use an SD memory card as an external storage to provide space for big data files.

While we have outlined the permission model, we also want to emphasize that some resources are freely accessible to every installed application. Certain system settings, such as volume or vibration settings, can be read and edited without any special permission [28]. Additionally, no permission is required for accessing information such as sensor data from accelerometers.

Overall, Android offers a wide range of features and resources. To protect the user, access is limited by predefined permissions, which are statically assigned to an application. The user has to make an informed decision whether to grant a certain set of permissions to an application or not. Furthermore Android only provides limited, well-defined IPC and otherwise aims at isolating the applications from each other.

2.2 Related Work

In this section we present the related work for this topic and take a look at previously described ideas and solutions concerning the confinement problem, covert channels and Android-specific research.

In 1973, Butler W. Lampson introduced the Confinement Problem [21] as the problem of limiting a running program from leaking data to third parties. Lampson mentioned examples including leaking data through IPC, the file system or file locks. Richard A. Kemmerer expanded on this and generalised the notion of covert channels [19] in order to accommodate storage and timing channels. He additionally presented a structured way to search for covert channels.

In connection with Android there have been numerous publications throughout the last years. Soundcomber [28] presented the concept of an Android banking trojan combining information extraction out of phone calls with communication through covert channels. This allowed low-privileged and apparently harmless applications to coordinate and have a powerful impact. Different covert channels on Android were presented. These channels were using different system states, such as screen, volume or vibration settings, as temporary data storage. We analyse some of these channels while also providing faster and stealthier channels.

Davi et al. [10] have demonstrated that the permission-model can also be bypassed through a chain of different attacks. They used low-level attacks, such as heap overflows, and made use of unprotected components of privileged applications, which allow unprivileged applications to escalate their privileges. This kind of transitive permission usage presents a serious threat, on which we will elaborate later.

There are approaches proposing to modify the permission-based security architecture slightly. Kirin [13, 14] was designed to check applications at install time and reject them if they match a security rule that indicates undesirably powerful applications. Alternatively there has been an approach to allow selective permissions [30] and thereby allow the user to install his desired applications, while preventing too powerful permission sets.

In order to prevent unauthorized and undetected data leakage on a smartphone, TaintDroid [12] was developed. It is based on information flow tracking and will be discussed in detail in Section 4.1. However, the effectiveness of taint tracking solutions for containment had been discussed before [8], leading to the conclusion that the evasion of taint tracking can be trivial and can appear in numerous different ways. On the other hand, previous work has also demonstrated how such trivial evasion could potentially be tackled [26]. We illustrate the feasibility of different channels out of the scope of TaintDroid.

Moreover, with XManDroid [4, 5] there is a security extension which tries to cover different channels and block communication that could lead to privilege escalation. We will discuss XManDroid in detail in Section 4.2.

Chapter 3

Channels

In this chapter we present different communication channels between two Android applications. We present *overt* and *covert* channels that we have implemented and will discuss their individual advantages and disadvantages. Additionally, we present a channel only requiring the installation of a single application and some ideas for future channels.

The channels pose a problem to the Android security model, as through their usage, applications can pool their privileges and thereby extend their available set of permissions. Colluding using such a channel allows two low-privileged applications to perform an operation, none of the two would have been able to perform individually.

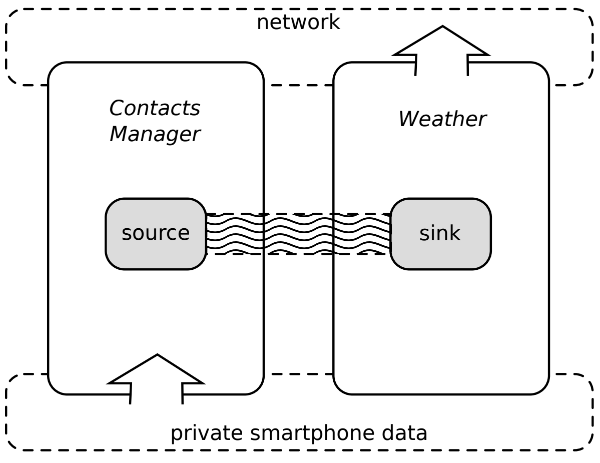


Figure 3.1: Illustration of an application collusion attack: as it has access to private data, the *Contacts Manager* acts as *source* and, as it can access the Internet, the *Weather* application acts as *sink*. The *source* sends a *message* to the *sink* through a channel [24].

3.1 Terminology

We will describe two different types of communication channels: *overt* and *covert* channels. As defined by Kemmerer [19], *overt* channels make use of data objects protected by access control, such as files or buffers. These data objects are intended to be used for data storage. On the other hand, data objects employed in *covert* channels were intentionally designed to store system state. However, by influencing the system state, such objects are subject to manipulation by processes running on the system and can therefore be used for communication.

As shown in Figure 3.1, we will describe channels allowing information flow from a *source* to a *sink*. If not stated otherwise, the *source* will be an application having access to sensitive information, but without outbound connection. The *sink* application however has an outbound connection while not having direct access to sensitive data. The transmitted data will be a *message* containing potentially sensitive information.

3.2 Prerequisites

Some of the channels require a shared secret between *source* and *sink*, such as a common meeting point or meeting time, in order to communicate. However, we feel that the search for such a shared secret in different applications can not be used as a detection method. Either the secret might be heavily obfuscated or the involved applications might generate it on the fly out of information that is available to all of them.

There are channels requiring a somewhat "quiet" environment or communication medium. We view this as realistic assumption. Applications can statically communicate at times that are likely to be "quiet", e.g. at night. Otherwise applications could dynamically analyse the users behaviour and initiate the communication, as soon as the conditions appear to be favorable. An example condition could be that the screen has been turned off for a substantial period of time.

For some channels it would be beneficial, if *source* and *sink* application would run under the same Unix user id. However, we experienced through the implementation that this is not a necessity and therefore assume the applications to have different user ids. We believe this improves the stealthiness of the channels.

To signal the end of the *message* to the receiver, most of the channels transmit a 0-byte. In case such a byte could appear in the communication, a proper encoding or a length field would have to be introduced in order to prevent data loss.

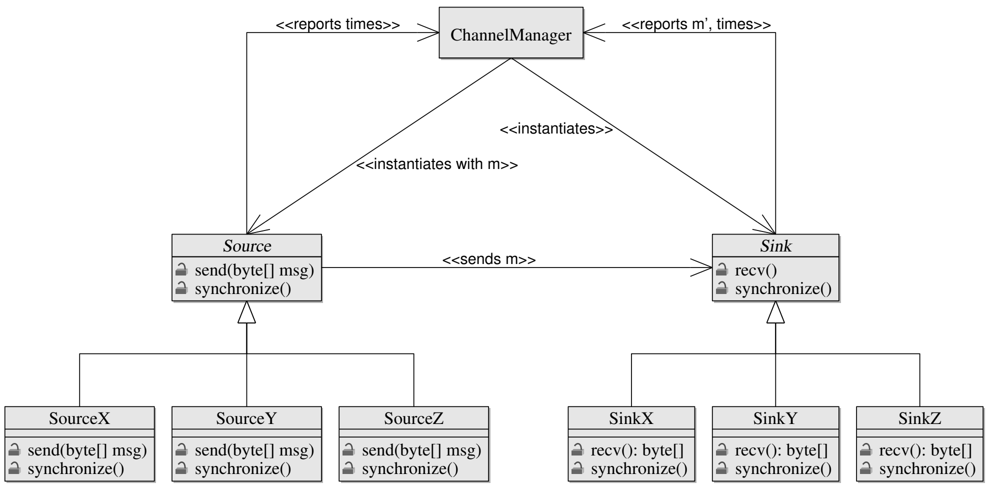


Figure 3.2: Overview of the used testing framework: The ChannelManager instantiates a *source* and a *sink* instance, the *source* sends the *message* m to the *sink* and the colluding applications report back to the ChannelManager for analysis.

3.3 Setup and Testing Conditions

We implemented the channels on two phones: Google Nexus One and Samsung Galaxy S. The Google Nexus One ran Android version 2.3.6, Codename Gingerbread, which at the time of this writing is the most widespread Android Platform Version[1]. The Samsung Galaxy S ran version 2.2.1, Codename Froyo, which had the second biggest market share. The respective API Levels were 10 and 8.

In order to test the functionality and characteristics of the channels, we implemented a testing framework, as shown in Figure 3.2. The framework allowed us to efficiently test the different channels using different configurations. The central component is the ChannelManager, which is started on the phone and coordinates all the actions. Depending on user input, the ChannelManager picks a channel, instantiates the *sink* and instantiates the *source* with a *message*. When started, *source* and *sink* synchronise and communicate while measuring their synchronisation times and communication times. The applications report back their results including the measured times and the received *message* to the ChannelManager. Finally the ChannelManager computes the bit-error percentage, saves all the information and either starts additional tests or notifies the user.

We send *messages* of sizes 4, 8 and 135 bytes, as test *messages*. They accordingly represented short pieces of information, GPS coordinates and six raw entries from the Contacts database with name and telephone number. When transmitting the information the applications were running in the

[1]http://developer.android.com/resources/dashboard/platform-versions.html

background and the screen as well as services, such as WiFi or Bluetooth, were turned off to reduce the general noise level.

In case the channel is based on a volatile data container, synchronisation is required in order to ensure that both applications are ready to communicate and none of the data is lost. The synchronisation time is measured from the point when both applications are running until they are synchronised in a way that they can begin communication. Although we did not perform optimisation on these times, we present the synchronisation times for completeness.

In order for the applications to continue running in the background, at least one of them has to acquire a wake lock[2], so that the CPU keeps running. In our tests we use a `PARTIAL_WAKE_LOCK`. The acquisition of a wake lock requires the `WAKE_LOCK` permission. Colluding applications could therefore either decide to acquire this permission or communicate at a different time, e.g. while charging, when the smartphone might keep running depending on the user's settings.

3.4 Overt Channels

In this section we present *overt* channels. They are not as well hidden as the later presented *covert* channels. The visibility of the *overt* channels could be limited by employing obfuscation techniques such as encoding.

The effectiveness of such obfuscation techniques can be limited and is subject to current research as we will outline in Chapter 4.

3.4.1 Channel Description

The descriptions of the channels in this section include their basic idea, their special characteristics as well as required permissions for the communication to work.

Communicating through External Storage

As users might want to store additional content such as music or video data on their smartphones, they have the possibility to use external storage. This channel makes use of such extra storage given two prerequisites are fulfilled. First, External Storage, e.g. an SD card, has to be available. Second, the SD card has to be writable by the *source*. This requires the *source* application to have the `WRITE_EXTERNAL_STORAGE` permission. However, this is a quite common permission, requested by a lot of applications.

[2] http://developer.android.com/reference/android/os/PowerManager.html

The channel itself is very simple. The *source* application writes to a predefined file in a public directory on the external storage and the *sink* application reads from this file. In our implementation we chose to write into the `download` directory. This appears relatively stealthy to us, as the user will probably use the `Downloads` application to browse this folder. The `Downloads` application however will only show actual downloads, so that the communication file remains invisible to the ordinary user. After a reasonable amount of time, the *source* could also delete the file again in order to lower the visibility of this channel.

This communication is clearly asynchronous and therefore does not require both applications to run at the same time. The implementation of the channel is fairly easy and big data chunks can be easily transferred.

Communicating through Internal Storage

The Android OS provides full access to a private directory on the internal storage for every application, which can be used to organize private data[3]. This allows the construction of a *covert* channel if the *source* application writes to a private file on its internal storage space and marks this file as world-readable. Given that the *sink* application knows the name of the *source* application, it can read from the world-readable file. In contrast to other Unix systems, the Android implementations we analysed did not offer a world-writable directory, such as `/tmp/`. We therefore we placed the file in a private directory.

This channel requires no permissions at all. As in the previously presented channel, the communication is asynchronous and straightforward to implement. However, the channel is less stealthy in the sense that it is relatively easy to detect the creation of a world-readable file in a private directory.

Using the System Log

Android offers a central Logging facility[4]. This is a useful development feature, which provides a simple way of logging with different verbosity levels. Android internally creates multiple different log files related to different contents. Applications that requested the `READ_LOGS` permission can read those log files. It has been shown before that these logs can contain lots of critical and private information [23]. Therefore, the read permission is also labeled as "dangerous" in the Android Reference[5].

[3] http://developer.android.com/guide/topics/data/data-storage.html#filesInternal

[4] http://developer.android.com/reference/android/util/Log.html

[5] http://developer.android.com/reference/android/Manifest.permission.html#READ_LOGS

To construct the channel, the *source* writes a specially marked log message. The *sink* application requests the READ_LOGS permission and parses the logs for such special messages. Parsing the logs is simplified, due to the fact that Android provides the logcat tool[6], which allows the specification of a message filter and can therefore be used to search for the special messages.

The logcat tool further allows every application to clear the entire logs. This can be used to hide the written messages towards other user applications once they have been read by the *sink*. However whether this increases the stealthiness or not is subject to debate, as a clearing of the log is also easily detectable and points to possible misuse.

More importantly, the *sink* has to read, while the special messages are still inside the log. The size of this time frame depends on the amount of messages other applications are logging, the size of the log as well as the appearance of log clearings. Therefore a "quiet" environment makes a lossless transmission easier to achieve. By repeatedly inserting messages over time, the *source* can increase the chances of the *sink* finding at least one of the messages.

The log is designed to handle ASCII data. Therefore non-ASCII *messages* have to be encoded, e.g. using Base64. The upper limit for a single line in the log is approximately 4000 characters. Therefore longer *messages* have to be split up into multiple fragments. Furthermore the log works as a ring buffer, which can cause trouble when sending *messages* bigger than the ring buffer size. Overall this channel requires a loose synchronisation as well as some implementation specifics, which make it harder to implement than the previously presented channels.

Shared Preferences

As a programming feature, the Android OS provides the ability to persistently store and manage key-value pairs as Shared Preferences[7]. These preferences can be created, accessed and modified through a simple API. When creating Shared Preferences, the creating application can set their operating mode. The operating mode can include MODE_WORLD_READABLE and MODE_WORLD_WRITABLE, which as they suggest allow other applications to read or write the Shared Preferences.

In order to allow an information flow the *sink* application creates a world-writable Shared Preference file for itself. The *source* application stores the data in a String object inside the created file. The *sink* can then fetch the data directly from its Shared Preferences. As with the previous Log channel, binary data should be encoded before transmission in order to allow a complete and correct transmission.

[6]http://developer.android.com/guide/developing/tools/logcat.html
[7]http://developer.android.com/guide/topics/data/data-storage.html#pref

This channel requires no extra permissions, allows asynchronous communication, the permanent storage of multiple *messages* and is easy to implement. However creating a world-writable Shared Preference can be detected rather easily. As the API creates and manages XML files in the applications internal storage directory, it would be possible to obfuscate this channel by replacing some of the API commands with standard file operations.

Using Broadcast Intents

We previously outlined how applications can perform inter-process communication using intents. This allows the creation of a trivial and straightforward channel. The *source* sends the data to the *sink* as part of an intent. Therefore it makes sense for the two applications to agree on an intent action. The *source* sets the action when creating the intent and the *sink* filters for the matching intents. This way the two applications can easily communicate.

The described procedure requires no extra permissions however the two applications have to be running at the same time, because the intent is volatile. To allow the *source* application to check, whether the *sink* is currently running as well, we implemented an acknowledgment sent back from the *sink* to the *source*. The acknowledgment uses the same technique and ensures the two communication partners of their presence.

The implementation of this channel is relatively straightforward, however its detection is easy, as the two applications communicate through the Android IPC mechanism, which can be filtered as seen in Section 4.2.

Using Unix Sockets

Employing native code, applications can efficiently communicate using Unix sockets. Unix sockets are a well-known method of inter-process communication allowing two processes to communicate in a server-client model. Unix sockets can either be bound to a file, be unnamed or use the abstract namespace[8]. For our implementation we chose the abstract namespace, because we can pick an address without requiring shared access to a file on the file system.

To allow communication, the *sink* opens up a socket in listening mode and the *source* connects to the listening socket. Afterwards the *source* can send data through the socket. The *sink* receives the data and might have to reassemble the fragments to restore the original data.

The communication is synchronous, but Unix sockets inherently contain ways to synchronise the two applications and check for the presence of the

[8] http://www.kernel.org/doc/man-pages/online/pages/man7/unix.7.html

Overt Channels	**Throughput (Kibit/s)**		**Bit Errors (%)**	
	Nexus One	Galaxy S	Nexus One	Galaxy S
Unix Sockets	340.45(± 154.02)	34.78(± 11.39)	0.00(± 0.00)	0.00(± 0.00)
Internal Storage	292.03(± 50.06)	32.60(± 8.47)	0.00(± 0.00)	0.00(± 0.00)
Shared Preferences	75.81(± 6.83)	31.00(± 2.75)	0.00(± 0.00)	0.00(± 0.00))
Broadcast Intents	40.58(± 8.41)	26.74(± 4.88)	0.00(± 0.00)	0.00(± 0.00))
External Storage †	11.55(± 1.10)	6.12(± 3.95)	0.00(± 0.00)	0.00(± 0.00))
System Log ‡	2.94(± 0.03)	2.14(± 0.11)	0.00(± 0.00)	0.00(± 0.00)

Overt Channels	**Synchronization (ms)**	
	Nexus One	Galaxy S
Unix Sockets	699.8(± 126.8)	441.0(± 186.3)
Internal Storage	N/A	
Shared Preferences	N/A	
Broadcast Intents	88.8(± 19.3)	61.8(± 26.4)
External Storage †	N/A	
System Log ‡	501.3(± 0.1)	516.3(± 4.2)

† Requires extra `WRITE_EXTERNAL_STORAGE` permission.
‡ Requires extra `READ_LOGS` permission.

Table 3.1: Listing of implemented *overt* channels with corresponding throughput in kibibit per second, bit-error percentage and synchronisation time (with the 95% confidence interval in parenthesis). The values shown are averaged over 10 runs, with 3 *messages* each, for both the Nexus One and the Samsung Galaxy S.

opposing side. The *sink* can wait for a new connection while the *source* can repeatedly try to connect until it is successful.

Because of the optimized and well-understood technology of this channel, it offers a very high bandwidth. However, the data is being openly communicated, which allows relatively easy detection.

3.4.2 Results

Table 3.1 shows the measurement results for the described *overt* channels. As expected, during the measurements of these channels no bit errors occurred. The table also gives the synchronisation times for channels, which require synchronisation. As the other channels use persistent storage and therefore can communicate asynchronously, they do not require synchronisation. As stated previously, synchronisation times are not optimized.

Considering the high throughputs of these channels, the biggest test *message* of 135 bytes is comparatively small. This results in relatively large variations and therefore larger confidence intervals. When tested with bigger *messages* these channels achieve throughputs, which have significantly higher orders of magnitude. However we present these results in order to keep the *overt* channels comparable to the later presented *covert* channels.

These channels clearly offer simple and reliable ways of communication. Their high throughputs allow the transmission of sensitive information of different sizes in sub-second time intervals. They are therefore very powerful and should be blocked.

3.5 Covert Channels

In this section we will outline the *covert* channels we implemented. In contrast to the previously explained *overt* channels, they rely on data containers that were not intended for communication. Therefore, they are generally harder to implement, but also harder to detect. We also discuss their performance when tested on the smartphones.

3.5.1 Channel Description

The channels described here are implemented in the previously described setup. As all channels do not store the *messages* persistently, they need a synchronisation mechanism. If not stated otherwise the synchronisation mechanism, uses the same technique as the channel and assures the two colluding applications of the presence of each other. None of the channels presented here requires additional permissions.

Using a Single Setting

As outlined, Android phones allow the user to customize a variety of settings. Amongst other things users can set the volume settings used when listening to music. Changing this particular setting requires no permission, while changing other settings require certain permissions[9].

Changing such a setting can be stealthy towards the user. If the setting is not used during the time of transmission, e.g. no music is played, the user does not notice it. Additionally, the original state should be restored after the communication has ended.

As any application can change this setting, it presents another possible data storage method and allows the construction of a channel [28]. During the communication, the setting is set to three different values: $0, 1, 2$. The values 0 and 1 are used to by the *source* signal the next bit to the *sink* and the value 2 is used by the *sink* to signal it has read the previous bit. Figure 3.3 shows an example of signaling the bits 1 and 0 to the *sink*.

This particular implementation requires the setting to allow at least three different values, but even with a binary setting such a channel is possible, when using a time-slotted protocol. However such an implementation is

[9]http://www.android-permissions.org/permissionmap.html

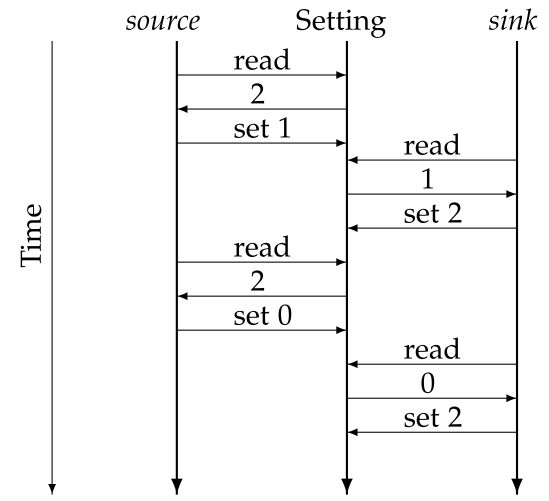


Figure 3.3: Single Setting channel transferring the bit sequence 1-0

slower and more error-prone, as the length of the time slot is difficult to choose. A shorter time slot increases the throughput, while also increasing the risk of bits being transferred incorrectly due to unfavourable scheduling.

As two applications are very frequently reading from and writing to a single setting, this channel can be detected by a supervisor. Such detection approaches will be discussed later. The possible throughput of this channel could be improved by incorporating more settings, as explained below.

Using Multiple Settings

This channel is very similar to the previously explained *Single Setting* channel. It uses system settings, which do not require permission to change them, to communicate. As an improvement this channel makes use of multiple different such settings at the same time.

In our implementation we used one vibration setting for synchronisation and multiple volume settings for parallel data transfer. This way two bytes are transferred after each synchronisation. As with the previous channel, it is only visible to the user if one of the settings is in use during communication, however it can be detected by a supervisor.

Using Automatic Broadcasts

Previous work [28] has shown that changing the vibration settings can also be used for creating a different kind of channel. Android has special settings, which, whenever changed, trigger an intent to be broadcasted through the system. The vibration settings are one of these settings[10].

Applications that have previously registered for this broadcast will receive an intent informing them about the new state of the vibration setting. Registration for this broadcast is available to any application, as it requires no specific permission. As the broadcast is basically the transmission of a single bit, it can be used to create a *covert* channel.

After the *sink* application has registered for these specific broadcast, the *source* simply changes the vibration settings according to the bit pattern of the *messages*. The *sink* receives the intents, decodes the bits and reconstruct the *message*. The Android OS takes care of the fact that the intents reach the *sink* in the correct order and delivers the intents rather quickly depending on the CPU power. Therefore the channel has rather high bandwidth, considering its single bit transmissions.

Intent Type

As intents provide an easy way for communication, we try to utilise them while not openly communicating data. Therefore, we encode the transmitted data in the additional fields of the intent, specifically the flags[11].

Additionally we could make use of other available intent features, such as the extras. Extras can store additional data and could be used in a *covert* way, e.g. by encoding information in the number of attached extras or by the existence or non-existence of certain extras.

Another available setting is the action field, which tells the intent receiver which action it shall perform. By using two actions such as `SaveZero` and `SaveOne` the information could be solely encoded in the sequence of received intent without the need to add any additional information.

Overall these channels are fairly easy to implement, as the communication uses the provided API. There is a vast variety of different possibilities for encoding the data, which makes it hard to detect what data is transmitted. However, depending on the amount of bits transmitted per intent, it is feasible to detect that the two applications are communicating as potentially many have to be exchanged.

[10] http://developer.android.com/reference/android/media/AudioManager.html#VIBRATE_SETTING_CHANGED_ACTION

[11] http://developer.android.com/reference/android/content/Intent.html#setFlags%28int%29

Unix Socket Discovery

This channel seeks to overcome the shortcomings of the previous channel using Unix sockets, while maintaining most of its benefits. It uses Unix sockets from the abstract namespace as well, however this time no data is transferred between the sockets. The *source* application acts as a server and sometimes provides, sometimes does not provide a listening socket. By trying to connect to the listening socket of the *source* and evaluating the result, the *sink* receives single bits. Whenever the connection attempt failed, the *sink* decodes this as a 0-Bit and as a 1-Bit upon successful connection establishment.

In order to synchronise the two sides, we use two additional, abstract Unix sockets. These sockets are used to signal the beginning and end of a single bit-transmission. The *source* application provides them to signal the information and the *sink* application repeatedly tries to connect until it is successful. As the primary communication socket, these synchronisation sockets do not receive any data, but communicate through successful connections.

This channel offers a relatively high bandwidth while its design still leaves room for improvement, e.g. by using multiple sockets for communication. Moreover this channel is also relatively *covert* in the sense that no data transfer is appearing.

Thread Enumeration

As on a regular GNU/Linux, `/proc/` holds information by the kernel about the system and the state of processes running on it. Inside `/proc/` each process has a dedicated directory. Amongst other things this directory contains the status file that provides insight into the memory usage, signal handling and the number of used threads.[12]

The status files are world-readable in the investigated Android versions and a process can "write" its status file by performing the corresponding operations. In our channel we combine these read and write operations to a communication channel based on two of the values inside the status file: Number of threads and locked memory size.

The *source* application creates and terminates dummy threads, in order to influence the number of threads. This value is read by the *sink* application, which derives the transmitted data from it. In order to achieve synchronisation between the two applications, they manipulate the locked memory size and thereby signal the other side their current state. This ensures that the number of threads is read if and only if the desired value has been reached.

[12] `http://www.kernel.org/doc/man-pages/online/pages/man5/proc.5.html`

After initial tests, we detected that there is a certain amount of noise on this channel. Initially the *source* application had nine threads. However one of them was periodically disappearing. In order to get a reasonable bandwidth and account for the noise, we do the following. We transmit four bits at a time. For the possible values between 0 and 15 the double amount of threads is created. This allows the *sink* to eliminate the noise of a single thread disappearing periodically.

The `status` file offers more information, such as additional memory usage information or signal bitmaps, which could be used to achieve a higher bandwidth. Additionally, parsing from the `stat` file, inside the process directory, can be more efficient. However the implemented channel demonstrates the problem of world-readable `/proc/` files. A detection of this channel would be feasible based on the observation that two applications are frequently reading each others `/proc/` files.

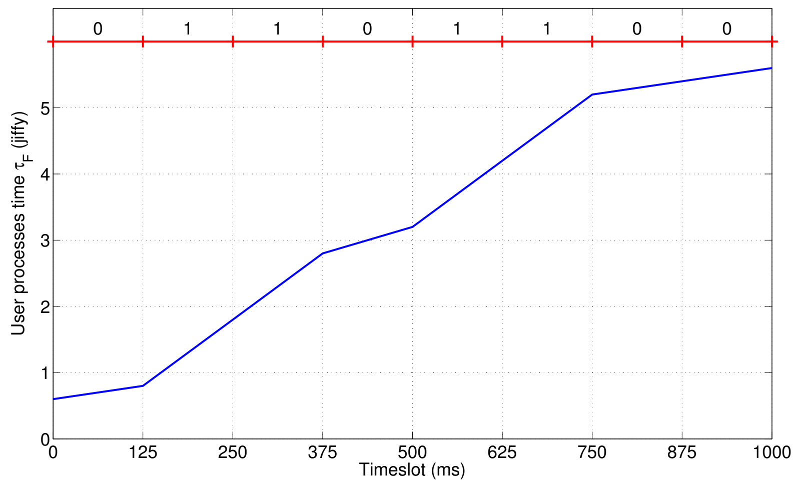


Figure 3.4: The schematic rise of the value inside `/proc/stat` when sending the bits, which are given at the top of the figure.

Reading /proc/stat

The previous channel has shown that reading from the `/proc/` directory can be beneficial when trying to create a *covert* channel. However in the previous example *source* and *sink* could be identified relatively easy when supervising `/proc/` directories. This channel also reads from `/proc/` but in a more *covert* way.

`/proc/stat` saves a number of kernel and system statistics, including the amount of time the system spent working on user processes[13]. This is a

[13]http://www.kernel.org/doc/man-pages/online/pages/man5/proc.5.html

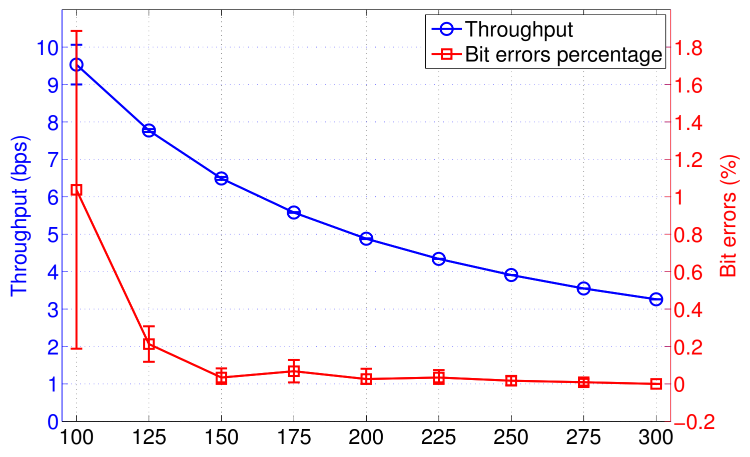


Figure 3.5: Illustrated trade-off between higher throughput and lower bit-error percentage by varying the length of the static time slot for the channel reading from `/proc/stat`. The amount of bit errors decreases with bigger time slots and vanishes for a time slot of 300 ms. These results were averaged over 10 runs, with three *messages* each, taken on the Samsung Galaxy S.

system-wide value and therefore does not directly link to a single process. As this value is monotonically increasing, user processes can only influence the amount by which the value increases.

A channel can be constructed from these starting points as follows. The colluding applications wait for a time with limited activity of user processes in order to limit the amount of noise introduced by other applications. The *source* application either tries to significantly increase the counter by performing intensive dummy operations or sleeps in order not to increase the counter, as seen in Figure 3.4.

We implemented this channel using fixed time slots. During a time slot the *source* performs one of the described actions depending on the current bit. The *sink* takes the value from `/proc/stat` at the beginning and end of the time slot and computes the difference. However, there is still a major problem to be solved. The *sink* has to define a threshold to decode the computed difference into a bit. As this threshold depends on the environment as well as the architecture, we implemented a learning phase, which allows the *sink* to find a usable threshold value.

During the learning phase, the *sink* will observe time slots with and without computation by the *source* application. We compute the threshold by averaging over the observed values. Using this threshold the two sides can perform the synchronisation and communication of the channel.

We believe that this channel makes it harder to identify the two colluding applications than the previous one. Especially the *source* application is hard to identify as it is simply performing dummy computations from time to time. However this channel is also significantly harder to implement. It requires a "quiet" environment in terms of user process activity, a learning phase to determine the correct threshold value and even afterwards is not safe from bit errors due to noise. Within certain bounds, throughput can be traded against bit-error rate by varying the static length of the time slot, as shown in Figure 3.5.

Free Blocks on the File System

For this *covert* channel we store information in a container that is accessible and mutable by both applications: the amount of free blocks on the file system. When the two colluding applications write data into their own private directories they use the same file system partition. By querying the number of free blocks, they can infer information about the file system usage of the other application. Out of this, we construct a channel. The *source* application will either reserve more space or free up space on the file system to signal a 1 or a 0.

When only the amount of free space is used as a communication mechanism several problems arise. First of all, there will probably be noise on this channel due to the fact that other applications also use the file system. In order to deal with the noise we use denoising on this channel. As the two applications communicate by the number of free blocks on the system, the *source* application will either reserve or free three blocks at once and the *sink* application will decode this using certain thresholds. Different denoising techniques could be applied, but we found these values to be a good trade-off between required time of the file system operation and the amount of noise that can be compensted.

A more general problem arises from the fact that this channel operates using fixed timeslots for synchronisation, as no other mechanism is available. Fixed timeslots however are conflicting with generally unbounded times for file system operations. Therefore the length of the time slot is a trade-off between throughput and possible bit errors, due to overlong file system operations that extend the time slot. If we want to ensure that our channel has a very low error rate, then we have to make sure that almost all execution times are within the time slot. This leads us to focus on the worst-case execution times.

The length of a proper time slot depends on the used file system and its characteristics. In our tests we experienced that the YAFFS2 file system on the Nexus One and the Robust FAT File System (RFS) on the Galaxy S were behaving significantly different. First of all their block sizes are 4 KiB and 16 KiB respectively. Therefore the channel on the Galaxy S has to write more data to the file system. With regards to execution times, our measurements have shown that the execution times on YAFFS2 are more predictable. On RFS however we observed file system operations, which took longer than 500ms. Such operations derail the synchronisation of the channel and cause bit errors.

Furthermore the proper length of the time slot also depends on the expected noise level induced by concurrent file system usage of other applications. These adjustments make the implementation rather difficult. However the resulting channel does not require any additional privileges and is very stealthy, as the *source* application just changes the size of a private file. This channel could be enhanced by combining it with another technique that provides synchronisation, as the currently used fixed timeslot is dependent on worst case runtimes.

Processor Frequency

Energy saving is an important issue on smartphones, but at the same time they are supposed to offer powerful performance when necessary. As a solution Android makes use of Dynamic Frequency Scaling (DFS), which is available in the Linux kernel. In DFS, depending on the current load pattern, the kernel decides how to change the clock frequency according to a selected algorithm: the frequency governor. The choice of a governor and its associated parameters determines how fast the kernel raises or lowers the frequency.

Using the fact that user processes can influence the clock frequency by inducing a high CPU load, we can construct a channel. While the sending application creates a load pattern according to the *message*, the receiving application observes the development of the processor frequency and reconstructs the bits.

Obtaining information about the current clock frequency can be performed in different ways, e.g. rather stealthy by repeatedly running a fixed number of operations and observing their execution times. In case of high clock rates execution times should be lower than in case of the lower clock rates. However, constructing such a channel is non-trivial as the two applications have to be tightly synchronised.

As such a tight synchronisation depending on different execution times and scheduling is hard to achieve, we decided to retrieve the current clock frequency in a faster and more reliable way: by querying it from the kernel by reading the value of the special file `/sys/devices/system/cpu/cpu0/-cpufreq/scaling_cur_freq`[14]. Through the use of multiple read operations we try to denoise the value in case of fast-changing governors and check whether it is ramping up over time, which would be the expected behaviour

[14] http://www.kernel.org/doc/Documentation/cpu-freq/user-guide.txt

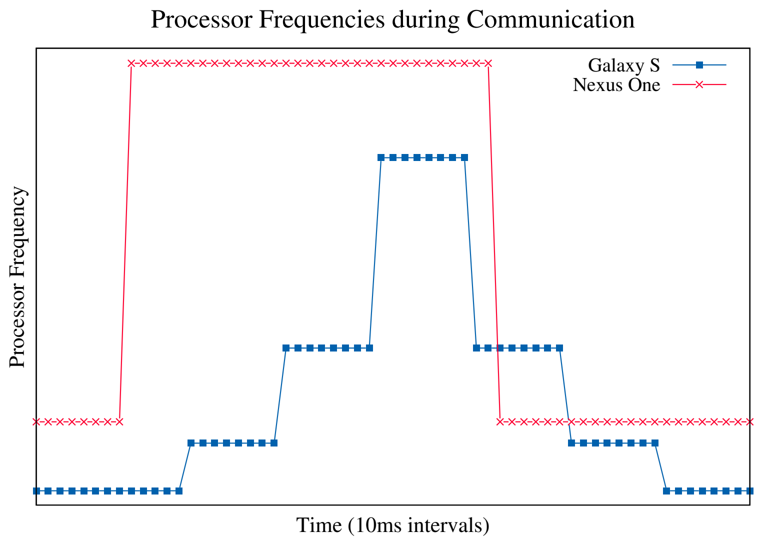


Figure 3.6: An example showing the different patterns in processor frequency caused by different frequency scaling governors, when creating 300ms of heavy CPU load.

in case the sending application intensively uses the CPU for an extended period of time. Fetching the current frequency value this way however is less stealthy than the previously presented way. For this channel we chose a simpler and more reliable method as its construction provides additional challenges.

The potential throughput of the channel depends on how well user processes can control the clock frequency. The faster the clock frequency changes the shorter the timeslot and the higher the potential throughput can be. This is because within a timeslot the processes have to wait until the desired state is reached and wait again until processor frequency returned to its original state. How fast the frequency can change depends on multiple factors. Most importantly the CPU frequency scaling governor and its parameters.

In our experiments we came across the *ondemand* on the Nexus One as well as the *conservative* frequency governor on the Galaxy S. As the names suggest the *ondemand* governor provides quick changes in the processor frequency, when demanded, while the *conservative* governor smoothly adjusts the processor frequency[15]. Additionally the chosen parameters influenced the governors in such a way that decisions about frequency changes where less frequent and less drastic for the *conservative* governor. An example can be seen in Figure 3.6, where the *ondemand* governor on the Nexus One reacts much faster and therefore allows a potentially higher throughput.

The creation of a channel is generally possible, however, the chosen fre-

[15] http://www.kernel.org/doc/Documentation/cpu-freq/governors.txt

quency governor and its options significantly influence the potential throughput and thereby the practical relevance. As seen, this channel is not easy to implement. It is stealthy except for the fact that the *sink* queries the current frequency many times. This channel is however particularly noise-prone and a "quiet" execution environment is required.

Timing Channel

Timing channels have been long known as a way of extracting information [20]. We present this *Timing Channel* that can be used for *covert* communication. As two colluding applications are running on the same phone, they share resources, such as the CPU. Different processes, which are competing for resources, interfere with each other and thereby influence each others execution behaviour. This allows a channel in which the presence and absence of interference can be used to transmit information. The sending application either does or does not perform some CPU-intensive operations. The receiving application always performs CPU-intensive operations and measures their runtime. Assuming roughly fair scheduling, the receiving application should be able to determine the *source* applications execution state.

In our implementation both applications perform dummy RC4[16] operations. After synchronisation, they communicate in fixed time slots and transmit one bit per slot. The *source* signals a 1 by doing computation in the slot and signals a 0 by sleeping. The *sink* goes through an initial learning phase in which it observes the spectrum of measured execution times in order to establish a threshold allowing the distinction between 1 and 0. As a threshold we use the moving average of previously observed execution times. After receiving enough measurements, the *sink* enters the synchronisation phase. Upon successful synchronisation the actual communication begins.

This channel is based on multiple assumptions. In order for the two applications to successfully communicate, they need relatively consistent channel behaviour. This includes that execution times throughout the communication time do not differ significantly. An especially negative impact is incurred by other applications that produce short, bursty CPU usage. These applications act similarly as the *source* application and can therefore cause the *sink* application to detect a 1, while a 0 was sent. Through our experiments, we found the Java Garbage Collection to have such CPU-usage patterns and cause the mentioned bit errors.

In order to avoid noise coming from processor frequency scaling, as explained for the *Processor Frequency* channel in Section 3.5.1, the two sides have to "wake up" the CPU. This ensures a certain processor frequency and

[16]http://www.openssl.org/docs/crypto/rc4.html

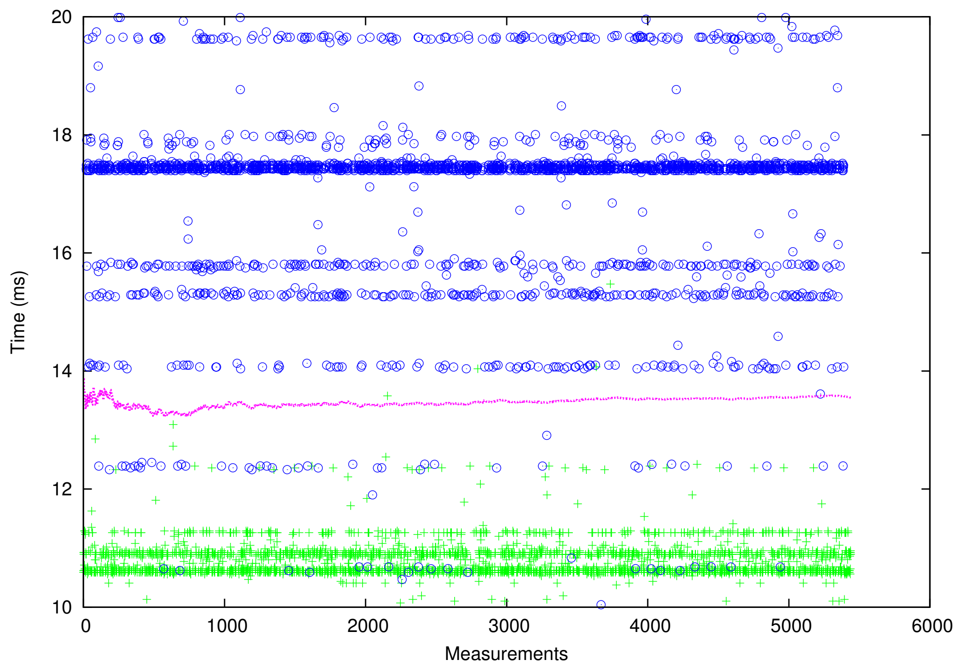


Figure 3.7: Measurements taken by the *sink* application to infer information sent over a Timing Channel. The blue dots show measurements while the *source* is sending a '1', green crosses while it is sending a '0'. 5400 measurements are needed to receive a 135-byte *message* (5 measurements are used to assign a value to a bit through a majority-vote mechanism). The red line shows a moving average used as a threshold value.

makes the channel more stable. To further denoise the channel, the *sink* takes multiple measurements and performs majority voting, as seen in Figure 3.7.

Once more the length of the time slot presents a difficult trade-off between throughput and bit-error rate. The result however is a very stealthy channel as all the applications do is perform computation. The necessary denoising techniques significantly lower the throughput, which therefore is lower than in previous channels.

3.5.2 Results

Table 3.2 shows the results for the different *covert* channels that have been tested in terms of throughput, bit errors and synchronisation time. On both of the phones, each of the channels has been run 10 times with three different *messages* transmitted in each run. We observe that the throughput of the different channels is in different orders of magnitude.

Based on the results, we observe two classes of channels here. The lower four channels of the table use static fixed time slots, while the other channels have more sophisticated mechanisms to keep them synchronised during

Covert Channels	**Throughput (bps)**		**Bit Errors (%)**	
	Nexus One	Galaxy S	Nexus One	Galaxy S
Type of Intents	3350.85(± 134.11)	4324.13(± 555.32)	0.00(± 0.00)	0.00(± 0.00)
Unix Socket Discovery	2610.92(± 305.25)	1647.78(± 170.70)	0.00(± 0.00)	0.00(± 0.00)
Multiple Settings	239.76(± 9.41)	284.91(± 1.90)	0.00(± 0.00)	0.00(± 0.00)
Thread Enumeration	157.73(± 0.97)	139.39(± 7.40)	0.00(± 0.00)	0.00(± 0.00)
Automatic Intents [28]	51.38(± 0.41)	90.67(± 0.39)	0.00(± 0.00)	0.00(± 0.00)
Single Settings [28]	46.88(± 0.31)	65.89(± 0.73)	0.00(± 0.00)	0.00(± 0.00)
Free File System Blocks	13.07(± 0.00)	9.80(± 0.00)	0.01(± 0.02)	0.03(± 0.03)
Reading `/proc/stat`	7.82(± 0.00)	3.26(± 0.00)	0.10(± 0.05)	0.00(± 0.00)
Processor Frequency	4.88(± 0.00)	0.47(± 0.09)	0.14(± 0.08)	4.67(± 2.26)
Timing Channel	3.70(± 0.00)	3.69(± 0.01)	0.10(± 0.11)	0.05(± 0.05)

Covert Channels	**Synchronization (ms)**	
	Nexus One	Galaxy S
Type of Intents	716.8(± 168.2)	473.0(± 249.0)
Unix Socket Discovery	5.2(± 0.8)	13.9(± 2.2)
Multiple Settings	314.9(± 21.8)	302.1(± 11.0)
Thread Enumeration	71.6(± 7.1)	110.1(± 8.8)
Automatic Intents [28]	1083.2(± 75.1)	435.1(± 180.8)
Single Settings [28]	267.5(± 3.2)	273.4(± 11.9)
Free File System Blocks	1038.2(± 5.1)	1442.7(± 15.6)
Reading `/proc/stat`	6923.4(± 8.1)	16669.2(± 48.7)
Processor Frequency	8203.9(± 7.2)	78866.1(± 9156.8)
Timing Channel	10286.8(± 16.1)	68057.6(± 105259.4)

Table 3.2: Listing of implemented *covert* channels with corresponding throughput, bit-error percentage and synchronisation time (with the 95% confidence interval in parenthesis). The values shown are averaged over 10 runs with 3 *messages* each for both the Nexus One and the Samsung Galaxy S.

communication. The lower four channels have lower throughputs as the applications always have to wait for the end of the slot and do not benefit from faster execution. Additionally, these channels have a certain amount of bit errors, some of which are caused by noise and some of which are caused by overly long execution times not fitting into the static time slot.

The *Processor Frequency* channel performs significantly worse on the Samsung Galaxy S because of its different frequency scaling governor, as outlined in the description of the channel. The less predictable behaviour leads to more bit errors and a lower throughput.

The channel using the *Free Blocks on the File System* performs worse on the Galaxy because of the reasons we outlined considering the different file system and its sometimes very slow performance.

Overall, we think that even the slower channels are still usable and therefore pose a threat to the security system. As an example, we show the times required for sending GPS coordinates (64 bit), including synchronisation time and sending the terminating null byte. Sending this data through the *Free Blocks* channel would take roughly 6.5 seconds on the Nexus One and roughly 8.8 seconds on the Galaxy S. The chances for error-free transmission would be above 99% and 98% for the two phones. From our perspective, this poses a relevant threat.

3.6 Collusion with Existing Applications

In this section we extend the concept of covert communication and make use of already existing applications as one of our colluding applications. This leaves us with two opportunities: either we replace the *source* or the *sink* with an existing application. Both of these options are feasible [23]. To replace the *source* we can find applications that have access to sensitive information and dump it into the system log. Therefore the *sink* from the *System Log* channel, described in Section 3.4.1, could be used to leak such data to the Internet.

However when we depend on another application as *source*, we can not control which data we will be able to read. Therefore, we will replace the *sink*, so that we keep control of the data selection [24].

3.6.1 Channel Description

In this section, we describe a channel that only requires a single installation and that we have successfully implemented and tested.

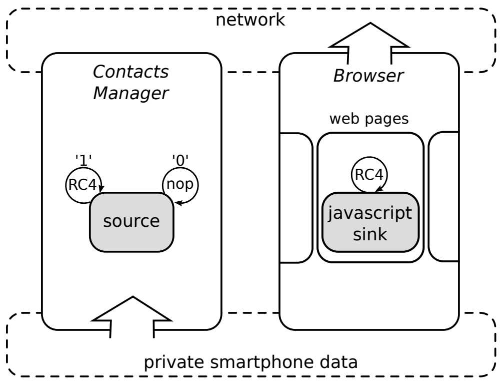


Figure 3.8: Collusion between a *source* application and a web script serving as *sink* [24]. The *source* uses RC4 operations to influence the processor frequency and the *sink* uses RC4 operations to estimate the frequency and infer the *message*.

Using the System Browser

In this scenario we replace the *sink* application with a piece of JavaScript that is embedded into a web page. The *source* application triggers the browser to open the web page at a time when the user is likely not to notice, e.g. at night, and tries to communicate with the JavaScript. As seen in Figure 3.8 the communication is made more difficult by an additional trust boundary as the browser executes JavaScript in a sandbox.

The channel we will construct is similar to the channel based on *Processor Frequency*, described in Section 3.5.1. However out of the JavaScript sandbox we can not query the processor frequency directly from the kernel. Therefore the channel works as follows: The *source* application manipulates the processor frequency according to the bit-pattern of the *message*. For a 1, it raises the processor frequency by performing dummy RC4 operations and for a 0 it sleeps to keep the processor frequency low. The *sink* tries to estimate the processor frequency and reconstructs the *message*.

To estimate the processor frequency, the *sink* tests how many, dummy JavaScript RC4[17] operations it can perform in a fixed time period. As explained, this requires tight synchronisation and is quite noise-prone. Therefore the colluding parties transmit each bit multiple times and the *sink* performs majority vote to determine the decoded bit.

The implementation of this channel is tricky. However, we think on the positive side it is also stealthy as long as the JavaScript can be hidden from the user. The communication can be hidden from the user by hiding the browser, redirecting to another page on completion or if the web page can be disguised as something desirable. For the browser to execute JavaScript, the screen apparently has to stay on. Therefore we kept the screen on using the according `WAKE_LOCK`. As discussed below the throughput is relatively low.

One-App Channel	**Throughput (bps)**		**Bit Errors (%)**	
	Nexus One	Galaxy S	Nexus One	Galaxy S
Browser Channel	1.29(± 0.00)	0.60(± 0.00)	0.43(± 0.61)	3.22(± 5.02)

One-App Channel	**Synchronization (ms)**	
	Nexus One	Galaxy S
Browser Channel	10268.0(± 35.0)	65083.0(± 60.2)

Table 3.3: Showing the implemented channel only requiring one application with corresponding throughput, bit-error percentage and synchronisation time (with the 95% confidence interval in parenthesis). The values shown are averaged over 10 runs with 3 *messages* each for both the Nexus One and the Samsung Galaxy S.

[17] http://code.google.com/p/crypto-js/

3.6.2 Results

The measurement results for the channel, which is colluding with an existing application, are shown in Table 3.3. As we can see, throughputs are very low. This is due to the necessary denoising techniques that require multiple transmissions. The Galaxy S is performing worse for this channel, as the channel is very similar to the channel based on processor frequency. For the *Processor Frequency* channel the Galaxy S was performing worse because of the *conservative* frequency governor, as explained in Section 3.5.1.

The transmission of data with 64 bits, such as GPS coordinates, would take around 66 seconds including synchronisation and termination. This might still be feasible is some scenarios.

3.7 Further Channels

Due to the limited time period we could not implement and test all the potential channels we identified. Therefore we list some more ideas here as future work.

As outlined by Murdoch et al. [25], as soon as two applications have general Internet access, they are able to communicate through different *covert* channels. However we do not see a reasonable scenario for collusion in this setup, as sensitive information could be sent anywhere directly. Therefore we do not cover these kinds of channels.

Analogous to the *Processor Frequency* channel, we described earlier, we could have created additional channels, based on kernel information. As an example a channel based on the amount of time the CPU was idle seems feasible. The kernel also provides such information[18].

As Lampson stated [21], the file system offers different opportunities for *covert* channels. Examples of how to use meta-data from the file system [27] can also be constructed on Android. Similarly to the *Socket Discovery* channel, explained in Section 3.5.1, we could implement a file discovery, which encodes information in the existence of a file.

As outlined previously it makes sense to combine some of the channels in order to eliminate their weaknesses. Namely a synchronisation method for the channel using the *Free Blocks on the File System* would be beneficial. Such a synchronisation could be provided by the *Timing Channel* in order to get a stealthy and better performing hybrid channel.

As we have explained, wake locks can be used to run applications in the background. However they could also be used to create a channel by encod-

[18] http://www.kernel.org/doc/Documentation/cpuidle/sysfs.txt

ing the *message* in the possibility to execute operations. Whenever the *source* holds a wake lock, it allows the *sink* to execute.

As the Android environment is so feature-rich, there are also other, more esoteric *covert* channels imaginable. We have previously mentioned that sensor data from accelerometers can be accessed without any permission. Our concepts test [16] on the Nexus One have shown that when is phone is lying on a solid ground, such as a tabletop, phone vibration is detectable by the accelerometer. This would allow channel using vibration to transmit information.

In another scenario the user could be actively used as a *covert* channel. The principles of such a channel has been demonstrated by TouchLogger [6], a key logger that uses the phone movements to infer the pressed keys. The user could be a channel when the *source* acts as a game that has to be solved by moving the phone according to a certain pattern and the *sink* observes the movements the user generates. Given that the user tries to win the game, the movement pattern should be quite significant and it might be possible to transfer information.

It has been demonstrated that information can be efficiently and loss-free encoded in sound signals [3], which also allows the creation of a *covert* channel on a smartphone. To communicate through this channel, the *source* plays specific sounds and the *sink* uses the microphone to receive and decode them.

Even if the last few ideas were more esoteric it gets obvious that a lot of very different *covert* channels seem possible, so that general solutions to block or detect *covert* channels on smartphones are unlikely to cover all the channels.

Chapter 4

Analysis of Existing Tools

In this chapter we present existing approaches aiming to prevent unauthorised data flows including sensitive information and evaluate their strength towards our previously described channels.

4.1 TaintDroid

Following the observation that a smartphone holds lots of sensitive information, which are handled by third-party applications, TaintDroid [12] tries to monitor the data flow of specified, sensitive data classes and alerts the user whenever it detects leakage of information from such classes.

4.1.1 Approach of TaintDroid

TaintDroid is implemented as a modification of the Android operating system. In order to alert the user, TaintDroid places taint sources in the system, which allow the initial identification of sensitive data. Such taint sources include private data, such as Contacts or stored SMS, sensor data from the camera or location information. Data retrieved from these sources is tagged and the tags are stored in shadow memory

Using dynamic taint-tracking, TaintDroid follows the information flow of the tagged data. According to a specified logic such tags are propagated or removed depending on the performed operations and their operands. The taint propagation is done on multiple levels. Inside the Dalvik VM TaintDroid employs variable tracking and is able to track taint propagation through primitive data types as well as exception handling or array lookups. Native code however is unmonitored by TaintDroid, so that it taints all values that have been accessed or are returned by native code. Additionally TaintDroid propagates Taint through IPC messages, by performing message-level taint tracking.

Whenever tagged data reaches the network as a taint sink, TaintDroid raises an alarm. The user is presented with a notification noting the application, which leaked the data, the class the leaked data originated from and the actual network transmission, as seen in Figure A.1.

4.1.2 Circumvention using Implicit Data Flows

As stated by the authors, TaintDroid does not track implicit data flows, but only explicit data flows. Implicit data flows can therefore be used to remove taint from tagged data and leak such data without raising an alert. Code 4.1 shows an example of an implicit data flow for a single bit between x and y. We present multiple relatively simple techniques that allow an adversary to remove the taint.

```
if( x ){
  y = 1;
}else{
  y = 0;
}
```

Code 4.1: Implicit data flow

Previous work has shown that such untainting can be a problem when trying to detect privacy leaks of untrusted code. If such code is aware of an environment, which tries to detect wrong-doing, then it is relatively easy to evade such detection techniques and very hard to counter the evasion, because a black-box approach would have to be applied [8].

A trivial n-way switch statement allows the untainting of $log_2(n)$ bits per execution of the switch statement [8]. An example for 2 bits is shown in Code 4.2. However through repeated execution long messages can be untainted. This method is very easy to implement, very fast and undetected by TaintDroid.

```
switch(x){
  case 0: y = 0; break;
  case 1: y = 1; break;
  case 2: y = 2; break;
  case 3: y = 3; break;
}
```

Code 4.2: Switch untainting

The Java Exception Handling offers another possibility to untaint data, as it allows a simple control flow redirection that can be used to construct an implicit data flow. As shown in Code 4.3, this method is also fairly easy to implement while offering a fast untainting procedure and being undetected by TaintDroid [26].

```
try{
  if( x ){
    throw new Exception();
  }
  y = 0;
}catch(Exception e){
  y = 1;
}
```

Code 4.3: Exception untainting

Additionally we present two new mechanisms that allow taint removal in the case of TaintDroid. First we leverage the file system and later we show the feasibility of having a timing-based taint removal.

As stated before, Android applications have read and write access to a private directory on the file system. This allows a procedure that utilises file creation and file checks to transmit and thereby untaint information. Code 4.4

demonstrates the procedure for transmitting a single bit. The information is encoded in the existence or non-existence of a private file with an arbitrary name. While this channel is also relatively easy to implement, its speed is limited by the operation times of the file system and the current file system usage.

```
if( taintedBit ){
    createLocalFile('untaint.txt')
}

if( fileExists('untaint.txt') ){
   untaintedBit = 1;
   deleteLocalFile('untaint.txt')
}else{
   untaintedBit = 0;
}
```

Code 4.4: Untainting using the file system

In order to demonstrate that also more advanced, hardly detectable techniques can be applied, we layout our implementation of a timing-based taint removal technique. It works very similar to the previously presented technique using the file system, except for the fact that for this channel the information is encoded in the execution time of a certain code section. As seen in Code 4.5 the application delays its own execution in order to transmit a 1. The achievable throughput of this channel is rather limited. It depends on the chosen delay. If the delay is chosen too short, the delay might appear through unfavourable scheduling, resulting in a flipped bit. This technique can be obfuscated further by the use of multiple threads or fake computation.

```
startTime = currentMilliSeconds();
if( taintedBit ){
    sleep(sleepTime);
}
endTime = currentMilliSeconds();

if( (endTime - startTime) >= sleepTime ){
   untaintedBit = 1;
}else{
   untaintedBit = 0;
}
```

Code 4.5: Timing-based untainting

Finally we present a table containing achieved throughput of our techniques, when removing the taint of random data blobs in Table 4.1. The tests were performed on a Nexus One with TaintDroid installed. Even though the time-based procedure is significantly slower we still consider it to be feasible and

Name	Throughput
256-way switch statement	~31,000,000 bps
Exception	~100,000 bps
File discovery	498 bps
Time-based	99 bps

Table 4.1: Untainting techniques tested on a Nexus One with the measured throughput when removing the taint off random blobs.

a threat to the system. Once an application is running it usually not in a rush to send off the data, as it can keep running in the background, and thereby has the required time for untainting. Additionally small data portions like GPS coordinates or credit card information can still be untainted in less than a second.

In general this taint removal can be viewed as a covert channel through which the application talks to itself. Therefore other mechanisms used for covert channels, which have been presented here or in previous work, could be used in order to perform untainting.

4.1.3 Native Code

During our tests, the TaintDroid modification did not allow JNI code execution from shared libraries contained in newly installed applications. Therefore we did not test further issues related to native code execution, which potentially would allow further ways of taint removal.

The policy of disallowing bundled libraries however seems very invasive, as libraries also appear in popular applications such as Angry Birds[1].

4.1.4 Taint Sink Testing

The TaintDroid authors claim that the network is used as a taint sink so that any tainted data being sent will raise an alarm [12]. Other taint sinks such as Bluetooth connections or sent SMS appear not to be used. Our verification tests revealed that sending contact information using the Java `HttpURLConnection` indeed triggered the alarm notification.

However sending the same data through a UDP connection using a Java `DatagramSocket` did not raise an alarm. Additionally the possibility to leak data through the browser, by opening a web site and providing confidential information as an argument, also went undetected [23]. We therefore conclude that the taint sink implementation of TaintDroid is incomplete and allows data leakage even without taint removal.

[1] `https://play.google.com/store/apps/details?id=com.rovio.angrybirds`

Name	Mean Bandwidth	Conf. Int.	Degradation
Internal Filesystem	48405.38 bps	13489.98 bps	83.8%
Shared Preferences	41309.36 bps	10552.54 bps	46.8%
Broadcast Intents	17541.89 bps	2888.21 bps	57.8%
External Filesystem†	8298.07 bps	1895.08 bps	29.9%
System Log‡	2538.80 bps	36.44 bps	15.6%
Type of Intents	1533.19 bps	96.18 bps	54.2%
Multiple Settings	225.53 bps	29.25 bps	5.9%
Automatic Intents [28]	45.61 bps	0.25 bps	11.2%
Single Setting [28]	40.31 bps	3.23 bps	14.0%

† Needs extra WRITE_EXTERNAL_STORAGE permission
‡ Needs extra READ_LOGS permission

Table 4.2: Throughput measurements on the Nexus One with TaintDroid installed. The throughput degradation is computed in comparison to the results without TaintDroid installed.

4.1.5 Detection of Existing Channels

When we tested our previously described channels on a Nexus One that had TaintDroid installed we made the following observations. The straightforward channels using the *Internal Storage* and the *Broadcast Intents* got detected. This was according to our expectations, as these channels did not hide their data leakage in any way.

Additionally the channel using *Multiple Settings* got detected. This was because we used byte-wise operations, such as bitmasks and shifts, to split up the data to fit it into the different settings and put it back together later. The taint propagated through all these operations.

The overt *External Storage* channel was not detected. As explained in the TaintDroid installation instructions[2] the external storage should be formatted as ext2 or ext3. However in our tests this channels was neither detected with FAT32 nor ext2.

The two other overt channels, *System Log* and *Shared Preferences*, are undetected by TaintDroid even though they should be detected. For the Log channel we found, that the log is written using native code, so that the taint can not propagate correctly. However we could not identify why Shared Preferences stays undetected. The covert channels were undetected as expected, as covert channels are not handled by TaintDroid.

All channels involving JNI Code execution could not be tested as they were blocked, as explained in Section 4.1.3. The three channels that initially got detected could evade detection by using one of the presented untainting techniques at the source. Table 4.2 presents an overview of the measurements with TaintDroid present and evasion techniques activated. We ob-

[2] http://appanalysis.org/download.html

serve that the channels perform worse because of the overhead induced by TaintDroid, however they are still applicable.

For all the channels tested in this scenario the bit-error rate was zero. Therefore we do not show this information in Table 4.2. As TaintDroid supervises accesses to the file system and additionally uses the file system for its own purposes, we can observe a degradation in the performance of channels using the file system. Additionally the monitoring of intents creates a certain overhead as mentioned by the authors, which shows in the reduced throughput of our intent-based channels.

4.1.6 Possible Improvements

The authors have stated that JNI code tracking and IPC tracking with a finer granularity might be future work. Additionally there has been work on how to identify implicit data flows [8]. However it is unclear to this point, whether this is practical on such resource-constrained devices.

4.1.7 Evaluation

Overall TaintDroid provides a suitable approach to detect unauthorised leakage of sensitive information sent over the Internet. However TaintDroid only detected two of the overt channels we implemented.

The TaintDroid authors have stated that they do not take implicit data flows into account. We feel that this is a serious limitation. A general TaintDroid evasion by taint removal is relatively easy and with acceptable or no degradation of throughput. The application author simply has to be aware of possible tracking techniques. Additionally TaintDroid can be evaded by taint sink avoidance. Therefore we conclude that the current TaintDroid implementation does provide neither a robust privacy protection nor a reliable detection of secret communication between applications.

4.2 XManDroid

XManDroid [4, 5] is a system trying to identify communication channels between applications in order to allow selective filtering to prevent confused deputy attacks and application collusion. In this section we describe XManDroid, analyse its performance when tested with our channels and conclude on its design.

4.2.1 Approach of XManDroid

XManDroid is an Android extension, which uses a system-centric approach in order to detect and filter communication between applications. XManDroid captures communication channels between applications and creates

```
<policy name="No leaking contacts">
  <node>
    <permission name="android.permission.READ_CONTACTS"/>
    <negpermission name="android.permission.INTERNET"/>
  </node>
  <node>
    <permission name="android.permission.INTERNET"/>
    <negpermission name="android.permission.READ_CONTACTS"/>
  </node>
</policy>
```

Figure 4.1: The XManDroid policy we used to test our channels. The *source* application that had access to the contact database is not allowed to share data with the *sink* application which can send data to the Internet.

a directed graph showing the information flow. Based on this global view, XManDroid allows the definition of policies, in order to block communication between applications having certain characteristics. These policies can be based on the permissions of the involved applications so that this procedure is not limited to specific applications but generally applicable.

The communication detection mechanisms of XManDroid work on multiple different levels. They capture communication through the file system, through sockets and through intents. Additionally XManDroid is able to detect information transfer through covert channels, created by repeatedly modifying system settings, as demonstrated in Section 3.5.1. In order to implement this TOMOYO Linux [18] is employed as it can intercept system calls and offers mandatory access control at kernel level. Whenever a communication link is identified, an according edge is added to the communication graph of applications.

Based on the current system state, given in the graph, and the policies, communication is either blocked or allowed. IPC calls are supervised using an extended Reference Monitor. These decisions are cached so that the overhead is relatively low and the authors claim it is not perceivable by the user.

In contrast to TaintDroid, XManDroid does not track individual information flows containing sensitive information, but observes communication channels between different applications and blocks communication which violates a policy not taking the actual communication content into account.

4.2.2 Detection of Existing Channels

Using our implemented channels we ran a series of tests on an early prototype of XManDroid that was installed on a Nexus One, running Android version 2.2.1 as provided by the XManDroid developers. We tried to leak contact information to the Internet by giving the source access to the contacts and the sink access to the Internet. We tested whether the "No leaking contacts" that is shown in Figure 4.1 would block communication between source and sink.

The prototype implementation of XManDroid that we could test blocked the communication for the overt *External Storage* channel and the covert channels *Single Setting*, *Multiple Settings* and *Automatic Intents*. These were channels XManDroid aimed at blocking so that these results were expected.

Because the available prototype came with a particular version of TOMOYO Linux, the overt channels *Internal Storage* and *Shared Preferences* were undetected. However the developers assured that these was a particular problem of the prototype, because the private directories had been white-listed. Additionally the channels *Broadcast Intents*, *Unix Socket Communication* and *Type of Intents* should be detectable by XManDroid, but are not detected by the prototype.

The channels using information out of `/proc/`, namely *Reading `/proc/stat`* and *Threads Enumeration* can be detected and blocked by XManDroid, because `/proc/` can be blocked by mandatory access control enforced by the TOMOYO kernel.

This leaves the channels *System Log*, *Unix Sockets Discovery*, *Free Blocks on File System*, *Processor Frequency* and *Timing Channel* uncovered. However the last two channels are low-level covert channels, which are not yet considered by XManDroid.

4.2.3 Evaluation

Overall XManDroid presents a promising and well-designed approach for the detection and prevention of covert communication. XManDroid is extensible in the sense that once a new detection mechanisms for covert communication has been found, it can be integrated into the existing framework. The additional edges are then appearing in the graph and the newly discovered covert communication can be blocked.

As limitation we would perceive the fact that XManDroid has to ensure it hooks into all the API functionalities that can be used to create a covert channel. Given the size of the Android API this is a challenging task. An additional limitation arises from the fact that the communication prevention appears regardless of the actual communication content and therefore might lead to false positives.

Chapter 5

Countermeasures

After stating a few general techniques usable against covert channels, we use this chapter to elaborate on some of the countermeasures that were discussed in previous work or that we newly introduce. We evaluate the applicability of countermeasures and state some that we would not implement.

5.1 General Techniques

There are multiple general techniques that can be applied as countermeasures [24]. We shortly outline them as we apply them later.

Limiting Multitasking: Multitasking is desirable from a usability view, however it enables all the channels that require synchronous communication. By more coarse-grained multitasking or limited multitasking channels can therefore either be blocked or seriously degraded.

Less powerful API: The more powerful the API is, the more effects and side-effects are resulting. Therefore, a more constrained API gives less opportunities for collusion and is easier to control.

More user interaction: An educated user might be able to adjust the system according to his wishes. However this is a very difficult trade-off as the user could also feel overwhelmed and ignore additional information.

5.2 Appropriate Countermeasures

In this section we present countermeasures that we view as appropriate in order to reduce the feasibility of application collusion attacks.

Limit System Log Readability

The Android System Log has been designed as a mechanism to simplify the development of applications on the Android platform. However more than once application developers have forgotten to strip all of their log messages before they deployed their applications. This allowed the access to a large number of critical information, as outlined in Section 3.4.1. Furthermore we have shown that a channel can be created through these log files, which is undetected by TaintDroid and XManDroid.

As it is very hard for users to understand the possible impact of this permission, we propose that access to the log is only granted, if the permission is granted and "USB debugging" is activated. This requires the user to enable "USB debugging" in the development settings, where he is prompted with a warning about the dangers to the integrity of his data. Furthermore the phone has to be connected to a computer through USB.

We think that this method would be very effective in blocking the problem, as ordinary users are unlikely to perform USB debugging. At the same time we believe that this method is not too invasive towards developers, as they probably read the log files through the `logcat` tool and the Android Debug Bridge(`adb`) on their computer at which time USB debugging is enabled anyway.

When analyzing 25900 random applications, we found that the `READ_LOGS` permission had been requested for 436 applications so that an estimated 1.68% of applications request it. Out of these 436 applications we found 38 that did not make use of the permission through static analysis [15]. These numbers are consistent with findings of the developers of XManDroid.

Overall we conclude that our proposal would block the misuse of a relatively powerful, harmless looking permission, which has not been used extensively. Furthermore this permission allows the creation of a channel, which is non-trivial to detect. Our proposal should not be too invasive as it still allows debugging for developers.

Selecting the Frequency Scaling Governor

In Section 3.5.1 and 3.6.1 we outlined the possibility to communicate using the processor frequency, which is enabled by the fact that user processes are able to control the frequency to a certain extent. We evaluate different strategies to counter this problem.

Dynamic Frequency Scaling can be disabled through the `performance` governor that sets the processor frequency statically to highest available one. Disabling frequency scaling completely obviously blocks the *Processor Frequency*away channel. However it will also increase the energy consumption of the phone, thereby lowering the battery-powered run time. Furthermore the *Timing Channel* and the *Reading `/proc/stat`* channel would be simplified, as they no longer have to take varying frequencies into account.

Our explanation as well as the results in Tables 3.2 and 3.3 demonstrated that the control by user processes is more limited in the case of a *conservative* governor, which was in use on the Galaxy S. This governor reacts slower and thereby makes it harder to control the frequency. This setting hinders multiple different channels and comes without negative impacts for the user.

We therefore propose to select the *conservative* governor for CPU Frequency Scaling as it makes communication significantly slower. Obviously this is just a complication as detecting or blocking such channels is considered very hard [11].

Limiting Access to `/proc/`

We outlined that the `/proc/` directory saves a lot of different status values, which allow a lot of different ways of covert communication. Previous work has noted that access to `/proc/` could be limited in order to disable or degrade some of the channels [24]. Different technical solutions exist as XManDroid made use of TOMOYO and another option would be the use of SEAndroid [29]. However there are different policies on how to restrict access to `/proc/`:

Totally blocking `/proc/` for all applications seems very restrictive. It is likely to disrupt applications that are checking `/proc/version` or `/proc/config` to obtain system information. Such applications could be system applications as well as some legitimate third-party applications, such as task managers.

Using a system, such as SEAndroid or TOMOYO, we can allow selective access to `/proc/`. We could allow applications to access the folder created for their uid, but prevent access from other directories. This would block channels such as *Thread Enumeration*. Additionally, we could allow access to general files in `/proc/`, which can not be influenced by an application, such as `/proc/version`. However we would deny access to files such as `/proc/stat` that depend on the behaviour of unprivileged applications. This policy blocks both `/proc/`-related channels we have presented, while we believe that it will not be too invasive.

The previous policies have not allowed the presence of a task manager that would need access to all directories. As long as such an application is present, we do not believe that blocking covert channels completely is possible. However, in order to prevent information leakage while allowing a task manager, we propose the following: every application having the `INTERNET` permission should not be allowed to read from `/proc/` directories other than its own. This would prevent possible covert channels through `/proc/` to be used as a way to leak data to the Internet. We believe that an efficient implementation would be possible as every application that has the `INTERNET` permission is already part of a special Unix group.

We have outlined that different policies exist in this domain. The decision is a trade-off between implementation complexity, usability and security. We would propose that selective access to `/proc/` is allowed. This blocks the described channels as well as possible side-channel attacks against other applications.

Restrict Access to the Browser

Any application can send an intent to the system browser in order to open a certain web page. This and the possible resulting issues had been presented in 2010 [23]. However as nothing changed, Thomas Cannon did an implementation, which demonstrated a remote-shell without any permissions [7]. This should clearly indicate that the right to command the browser can result in a full Internet access. In order to remove this shortcoming we outline possible solutions.

As opening the browser can lead to full Internet access, the Browser can only be opened by applications that themselves have the `INTERNET` permission. This proposal is rather restrictive as the possibility to open the browser was put in place especially for applications that do not require full Internet access. These applications had the chance to forward users to web sites or show advertisements.

As an alternative, opening the browser with the screen turned off could be denied in order to protect the user from attacks similar to the one by Thomas Cannon. However this would still leave room for other attacks.

Finally an increased amount of user interaction could help to prevent the problem. Either the user would be asked whenever an application tries to open the browser or a system-wide preference could be chosen. However for two reasons this does not seem like an appropriate solution to us. The ordinary user would have trouble deciding on whether or not to open a certain web site. Additionally the user is left with almost no time to correct his decision as data can be leaked very quickly.

Overall blocking requests to open the browser from applications that do not have Internet access seems to be the best solution to us. It is rather invasive but also tackles a very serious problem.

Selective Fuzzy Timing

As described, hardware-level channels, such as *Timing Channel* or *Processor Frequency*, are very hard to prevent. In order to block them on Android, we propose the following: applications that require precise timing, have to request such timing using a new `REQUIRE_PRECISE_TIMING` permission. Such applications are then handled in a special way.

As we have seen the hardware-channels already have a relatively low bandwidth. In order to work properly, they have to be well synchronised and therefore require precise timing. If such precise timing is not present these channels either fail or are unusable because of their extremely low throughput. Test measurements with our working Timing Channel on the Nexus One have shown that fuzzy timing that only moves in 20 ms intervals is sufficient to completely block communication. Of course the parameters of fuzzy timing, sufficient to block communication, are dependent on the implementation. However we believe that there exists a value, sufficiently small so that normal applications are not affected and sufficiently large to make hardware channels infeasible.

If two applications having the new `REQUIRE_PRECISE_TIMING` permission would run in parallel at least one of them has to be running in the background. We can think of different possible policies to block communication in this case. Either we use CPU slicing to limit the affect the applications can have on each other or we pause the background application. The Android reference explicitly states that background services have to be prepared to be interrupted[1].

This new permission would allow the introduction of fuzzy timing and would allow automatic actions by the system that are intended to render communication on a hardware-level unusable, because their bandwidth is too low.

5.3 Inappropriate Adjustments

As stated at the beginning of this thesis Android already contains a very wide range of different permissions. Furthermore we require that the user has a certain understanding of the permissions and their impact, so that he can make an educated decision whether or not he wants to install the application. We therefore believe that the introduction of new permissions should only occur if they are easily understandable and allow the system to block previously possible wrong-doing.

As an example, we do not believe that splitting up or limiting the very powerful `INTERNET` permission is a good idea. Possible solutions could be additional white lists for accessible URL per application or differentiating depending on the type of traffic. This would generally not prevent informa-

[1]http://developer.android.com/reference/android/app/Service.html#ProcessLifecycle

tion leakage as covert communication on network layer is possible [25] and as collusion partners might reside in the networks or at one of the endpoints.

Another example would be the addition of a permission to change the volume and vibration settings. This could prevent the channels introduced by Soundcomber [28]. However as a lot of application might request such a permission this would bloat the lists of required permissions and would likely lower the user's overall permission awareness.

5.4 Final Remarks

We have seen that certain hardware-level channels are very hard if not impossible to close completely. Probably no single countermeasure will be able to shut down covert channels as a whole. Additionally poor programming of privileged applications can lead to a transitive permission misuse allowing covert channels through these applications. This problem arises when powerful interface are not or only poorly protected [10]. Such design errors are very hard to be handled correctly by possible countermeasures as the distinction between legitimate and illegitimate use is usually non-trivial.

Chapter 6

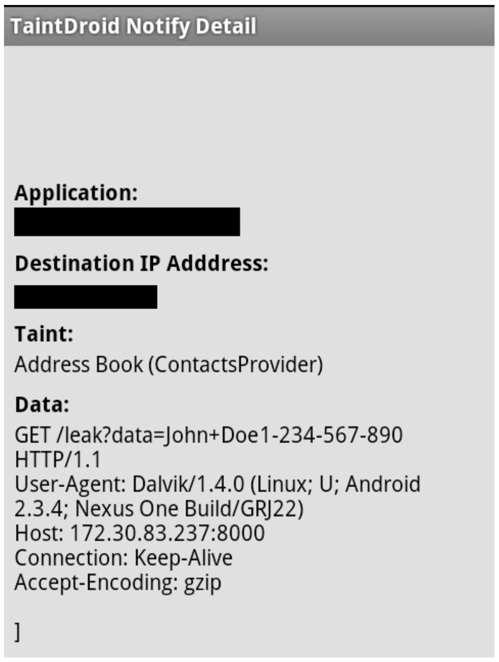
Conclusion

In this thesis, we have implemented and analysed a variety of overt and covert channels between Android applications. The functionalities of these channels range from being hardware-related, for the *Timing Channel*, over being OS-related, for *Unix Socket Discovery* or *Thread Enumeration*, to being high-level API-related, for *Single Setting* or *Type of Intents*. We have shown that, due to their fundamental differences, it is very hard if not impossible for a single countermeasure to block or detect all the channels. We have analysed the existing tools TaintDroid and XManDroid, which we showed were only able to detect or block a subset of our presented channels. Therefore, application collusion attacks remain an open research problem. We have then proposed a number of different countermeasures, which tackle some previously unsolved problems. As part of an extension to the typical application collusion problem, we have demonstrated the possibility for covert collusion with an existing application, which reduces the requirements to mount a collusion attack as only a single application must be installed.

While stealthy communication on Android devices remains an open problem, we have demonstrated the benefits and limitations of existing countermeasures as well as our own countermeasures. Given the implications of hardware-related channels, i.e., their throughput which is sufficient to transfer private information, application collusion attacks remain an interesting research problem.

Appendix A

Appendix



Application:
[redacted]
Destination IP Adddress:
[redacted]
Taint:
Address Book (ContactsProvider)
Data:

```
GET /leak?data=John+Doe1-234-567-890
HTTP/1.1
User-Agent: Dalvik/1.4.0 (Linux; U; Android
2.3.4; Nexus One Build/GRJ22)
Host: 172.30.83.237:8000
Connection: Keep-Alive
Accept-Encoding: gzip

]
```

Figure A.1: Sample Notification of TaintDroid when an application tries to leak the Contact information for the sample contact John Doe.

Bibliography

[1] Google Android. http://www.android.com/.

[2] Google play. http://play.google.com/about/features/.

[3] Iftach Ian Amit. Data Exfiltration - the way Q would have done it. Technical report, SOURCE Barcelona, 2011.

[4] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks. Technical Report TR-2011-04, Technische Universität Darmstadt, Apr 2011.

[5] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards Taming Privilege-Escalation Attacks on Android. In *19th Annual Network & Distributed System Security Symposium (NDSS)*, Feb 2012.

[6] Liang Cai and Hao Chen. TouchLogger: Inferring Keystrokes On Touch Screen From Smartph one Motion. In *6th USENIX Workshop on Hot Topics in Security (HotSec 11)*, San Francisco, CA, August 2011.

[7] Thomas Cannon. Android No-Permissions Reverse Shell. http://vimeo.com/33576202.

[8] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. On the Limits of Information Flow Techniques for Malware Analysis and Containment. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, July 2008.

[9] comparis.ch AG. (2,9 Millionen Schweizer haben ein Smartphone). http://www.comparis.ch/~/media/files/mediencorner/medienmitteilungen/2012/telecom/verbreitung-smartphone.pdf.

[10] Lucas Davi, Alexandra Dmitrienko, Ahmad-reza Sadeghi, and Marcel

Winandy. Privilege Escalation Attacks on Android. *Information Security*, 6531:346–360, 2011.

[11] Dorothy E. Denning and Peter J. Denning. Data Security. *ACM Computing Surveys (CSUR)*, 11:227–249, September 1979.

[12] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 393–407, Vancouver, October 2010.

[13] William Enck, Damien Octeau, Patrick Mcdaniel, and Swarat Chaudhuri. A Study of Android Application Security. *USENIX Security*, (August):935–936, 2011.

[14] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. *Security*, pages 235–245, 2009.

[15] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[16] William J. Francis. A quick tutorial on coding Android's accelerometer. http://www.techrepublic.com/blog/app-builder/a-quick-tutorial-on-coding-androids-accelerometer/472.

[17] Inc. Gartner. Gartner Says Worldwide Smartphone Sales Soared in Fourth Quarter of 2011 With 47 Percent Growth, 2012. http://www.gartner.com/it/page.jsp?id=1924314.

[18] T. Harada, T. Horie, and K. Tanaka. Task Oriented Management Obviates Your Onus on Linux (TOMOYO Linux). Linux Conference, 2004.

[19] Richard A. Kemmerer. A Practical Approach to Identifying Storage and Timing Channels: Twenty Years Later. In *18th Annual Computer Security Applications Conference (ACSAC)*, pages 109–118, December 2002.

[20] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*, pages 104–113, 1996.

[21] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16:613–615, October 1973.

[22] Zach Lanier and Jon Oberheide. TEAM JOCH Presents: Lessons In Mobile Penetration Testing, 2011. SOURCE Barcelona.

[23] Anthony Lineberry, David Luke Richardson, and Tim Wyatt. These aren't the permissions you're looking for. Technical report, Blackhat, 2010.

[24] Claudio Marforio, Francillon Aurélien, and Srdjan Čapkun. Application Collusion Attack on the Permission-Based Security Model and its Implications for Modern Smartphone Systems. Technical Report 724, ETH Zurich, April 2011.

[25] Steven J. Murdoch and Stephen Lewis. Embedding Covert Channels into TCP/IP. In *Information Hiding*, pages 247–261, 2005.

[26] Srijith K. Nair, Patrick N. D. Simpson, Bruno Crispo, and Andrew S. Tanenbaum. A Virtual Machine Based Information Flow Control System for Policy Enforcement. *Electron. Notes Theor. Comput. Sci.*, 197(1):3–16, February 2008.

[27] P.A. Porras and R.A. Kemmerer. Covert flow trees: a technique for identifying and analyzing covert storage channels. In *Research in Security and Privacy, 1991. Proceedings., 1991 IEEE Computer Society Symposium on*, pages 36 –51, may 1991.

[28] Roman Schlegel, Kehuan Zhang, Xiaoyong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*, pages 17–33, February 2011.

[29] Stephen Smalley, NSA, and Trust Mechanisms (R2X). SEAndroid. `http://selinuxproject.org/page/SEAndroid`.

[30] Whisper Systems. Selective permissions for Android. `http://www.whispersys.com/permissions.html`.

Date Internet sources have been accessed: Friday 27th April, 2012