Analysis of the Communication between Colluding Applications on Modern Smartphones

Claudio Marforio[†], Hubert Ritzdorf[†],
Aurélien Francillon[‡], Srdjan Capkun[†]

[†]Institute of Information Security
ETH Zurich, Switzerland
{maclaudi,rihubert,capkuns}@inf.ethz.ch

[‡]Networking and Security Group
Eurecom, Sophia-Antipolis, France
aurelien.francillon@eurecom.fr

ABSTRACT

Modern smartphones that implement permission-based security mechanisms suffer from attacks by colluding applications. Users are not made aware of possible implications of application collusion attacks—quite the contrary—on existing platforms, users are implicitly led to believe that by approving the installation of each application independently, they can limit the damage that an application can cause.

We implement and analyze a number of covert and overt communication channels that enable applications to collude and therefore indirectly escalate their permissions. Furthermore, we present and implement a covert channel between an installed application and a web page loaded in the system browser. We measure the throughput of all these channels as well as their bit-error rate and required synchronization for successful data transmission. The measured throughput of covert channels ranges from 3.7 bps to 3.27 kbps on a Nexus One phone and from 0.47 bps to 4.22 kbps on a Samsung Galaxy S phone; such throughputs are sufficient to efficiently exchange users' sensitive information (e.g., GPS coordinates or contacts). We test two popular research tools that track information flow or detect communication channels on mobile platforms, and confirm that even if they detect some channels, they still do not detect all the channels and therefore fail to fully prevent application collusion. Attacks using covert communication channels remain, therefore, a real threat to smartphone security and an open problem for the research community.

1. INTRODUCTION

Modern smartphone operating systems allow users to install third-party applications directly from on-line *application markets*. Given the large number of applications and the number of independent developers, every application cannot be trusted to behave according to its declared purpose. Certain types of malicious behaviors can be detected by inspection and testing while others cannot; malicious applications therefore find their way into application markets [1, 24, 25, 26].

To limit the potential impact of malicious applications, mobile phone operating systems (e.g., Android OS [14], Symbian OS [23], Windows Phone 7 [21]) implement a permission-based security model (also called a *capability model*) that restricts the operations that each application can perform. This model explicitly gives applications the permissions that are required to correctly execute their operations. Recent work by Schlegel et al. [29] introduces smartphone malware that makes use of application collusion over a limited number of communication channels to overcome the security mechanisms put in place by the implemented permission-based model. By establishing communication over a covert or overt channel, applications are allowed to execute operations which the system, based on their declared permissions, should prohibit.

It is important to stress that application collusion attacks on permission-based models are neither a result of a software vulnerability nor related to a particular implementation. Instead, they are a consequence of the basic assumption on which the permission-based model relies: applications can be independently restricted in accessing resources and then safely composed on a single platform. Collusion attacks show that this assumption is incorrect and can be exploited to circumvent the permission-based model. Furthermore, in current systems, users are not made aware of possible implications of application collusion attacks—quite the contrary—users are implicitly led to believe that by approving the installation of individual applications independently, they can limit the damage that an application can cause.

Although the existence of overt and covert channels and thus the feasibility of application collusion on any platform might not be surprising, the implications of collusion are very damaging on mobile platforms: these platforms are designed for personal use, generally store personal information and facilitate the installation of multiple third-party applications. Furthermore, existing security products (e.g., Lookout Privacy Advisor [32]) analyze and report application permissions independently for each individual application. Since they do not consider application collusion, these products do not correctly reflect the collective privacy implications of the applications that the users install.

In this work, we demonstrate the existence of a number of overt and covert channels by implementing them on the

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

Android platform. We then evaluate the severity of the threats posed by application collusion attacks by measuring the throughput and stability of each channel. Our results show that different covert channels, which are generally hard to detect or prevent, can reach throughputs roughly ranging from 3.7 bps up to 3.27 kbps on the Nexus One and from 0.47 bps up to 4.22 kbps on the Samsung Galaxy S, thus posing a serious threat to privacy on modern smartphones.

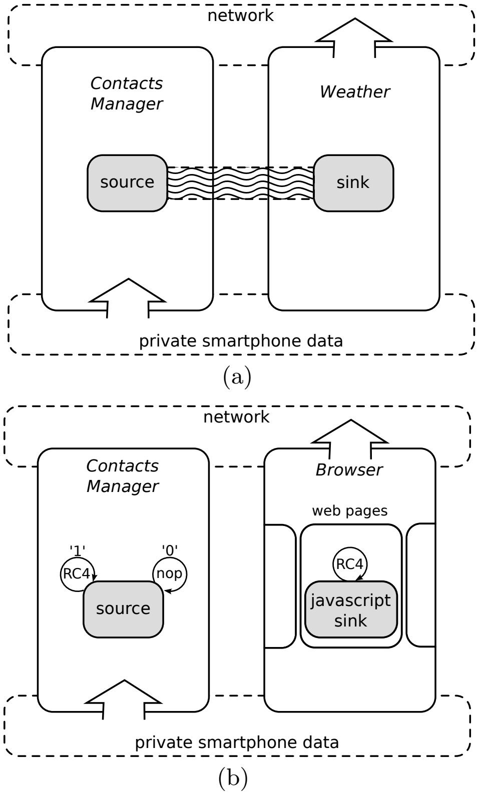
While it is shown that overt channels on mobile smartphones may be detected or restricted using taint analysis [10], policy enforcement [3, 4], better sandboxing and by reducing access to some APIs, we show that these approaches fail to detect most covert channels. This is consistent with research carried out in the 1970's where it has been shown that covert channels in computer systems are hard to prevent [9, 20]. To evaluate the effectiveness of contemporary tools, we tested both TaintDroid [10] and XManDroid [4] confirming that they do not detect all channels and therefore fail to fully prevent application collusion. This shows that application collusion attacks remain a real threat on modern smartphone platforms. Finally, we propose ways of preventing or limiting some of these channels.

In summary, the contributions of this paper are the following. (1) We demonstrate the practicality of application collusion attacks by implementing several communication channels on the Android platform. (2) We measure and report the throughput of implemented communication channels highlighting the extent of the threat posed by such attacks. (3) We confirm that two recently proposed architecture modifications and tools that deal with overt and covert channels discovery, TaintDroid [10] and XManDroid [4], still fail to detect several of the implemented channels. (4) We propose countermeasures that, if not eliminate, limit the throughput of selected communication channels between the applications.

The rest of the paper is organized as follows. In Section 2 we present the problem statement with a classification of communication channels. We then show the results of our study of communication channels in the Android OS in Section 3. The analysis of TaintDroid and XManDroid in the setting of our testing framework is detailed in Section 4. We then discuss current mitigation techniques and their limitations in Section 5. Finally, we present related work in Section 6 and conclude in Section 7.

2. PROBLEM STATEMENT

The goal of this work is to understand the threat posed by colluding applications on modern smartphones. In particular we investigate the feasibility and the practicality of multiple communication channels in terms of throughput, bit-error rate and required synchronization. Figure 1(a) illustrates an example channel between two applications. On the left, the *ContactsManager* application has access to private data on the device but not to the network (later in the work referred to as the *source* application), on the right, the *Weather* application having access to the network but no direct access to the private data (later in the work referred to as the *sink* application). The two applications can create a stealthy communication channel to share data. Figure 1(b) illustrates an interesting covert channel that can be created between an application and the Browser which does not require any extra application to be installed on the device. We will describe this channel in more detail in Section 3.3.



(a)

(b)

Figure 1: Figure (a) shows a generic example of collusion between the *ContactsManager* and the *Weather* applications through a stealthy (overt) communication channel. Figure (b) shows a (covert) timing channel between an application and the browser working through the use of dummy RC4 operations.

2.1 Channels Classification

We classify communication channels based on their implementation on current smartphone architectures as follows:

- **Application.** This is the level of the API that an operating system provides to the developers (e.g., Android's Java API, Windows Phone 7 C # / Silverlight APIs, iOS's Objective-C API). Access and usage of these communication channels may be easily controlled. We consider these channels as *high*-level.

- **OS.** This is the level of the operating system that is exposed through native calls that exploit information present in the operating system. We believe that at this level some channels are impossible to close, others, if closed, could hamper backward compatibility severely.

- **Hardware.** This is the level that is exposed through exploiting hardware functionalities of the smartphone. It is highly dependent on individual hardware specifications of smartphone models. These communication channels cannot be closed without severe performance degradation of the system. We consider these channels as *low*-level.

Different levels usually also imply different throughput and stealth. In particular, we notice that throughput is usu-

Overt Channel	**Throughput (kbps)**			
	Nexus One		Samsung Galaxy S	
UNIX Socket Communication	340.45	(± 154.02)	34.78	(± 11.39)
Internal Storage	292.03	(± 50.06)	32.60	(± 8.47)
Shared Preferences	75.81	(± 6.83)	31.00	(± 2.75)
Broadcast Intents	40.58	(± 8.41)	26.74	(± 4.88)
External Storage †	11.55	(± 1.10)	6.12	(± 3.95)
System Log ‡	2.94	(± 0.03)	2.14	(± 0.11)

† Requires extra `WRITE_EXTERNAL_STORAGE` permission.

‡ Requires extra `READ_LOGS` permission.

Table 1: List of our implemented *overt* channels in the Android OS with corresponding throughputs (with the 95% confidence intervals in parenthesis). The displayed values are averaged over 10 runs for both the Nexus One and the Samsung Galaxy S. The "System Log" is a new channel that we engineered and for which we did not find references in the open literature.

ally directly proportional to the level, with higher throughput associated to *high*-level communication channels. Stealth (i.e., the difficulty to detect a communication channel), on the other hand, is usually inversely proportional to the level, with stealthier channels associated to *low*-level communication channels.

3. OVERT AND COVERT CHANNELS IN ANDROID

We explore possible covert and overt channels on Android smartphones. We analyzed some known channels and identified a number of new channels specific to smartphones not yet presented in the literature.

To analyze overt and covert channels, we implemented a framework to measure the throughput, the bit error rates and the synchronization times for each implemented communication channel. The results of our study are presented in Tables 1 and 2.

The values shown in both tables are averaged over 10 independent runs for each implemented channel executing on a Nexus One or a Samsung Galaxy S smartphone. During the tests the phone was running on battery power and not charging. Each time the *source* application tries to send 4, 8 and 135 byte (to mimic the transfer of contact information as explained in Section 3.4) messages to the *sink* that, if the channel is open, would record them successfully. For each covert channel that requires tight synchronization between the *source* and the *sink* application (i.e., timing channels), we implement a synchronization protocol and run it before starting to send data. In general the synchronization protocol is used to estimate the noise present in the system when the applications want to share data and to correctly start the measurements to exchange data at the same time. Such a protocol is implemented on top of the covert channel (i.e., using the same mechanism) and allows us to reach higher accuracy. A covert channel's accuracy is measured in bit error rate, with perfectly accurate channels reaching 0% bit error rate during transmission. While synchronization time values are reported in Table 2 for completeness, we believe that they can be further optimized to yield overall slightly faster communication channels.

3.1 Overt Channels

We now briefly describe the implementation of overt channels to give an intuition of how they work.

Shared Preferences (Application): the *sink* application uses an API to create an Android preference XML file that is world-readable and world-writable. The *source* application writes ASCII data to it and the *sink* reads it. This channel does not require any synchronization to operate as the two applications do not need to be run simultaneously.

Internal Storage (Application): the *source* application writes a world-readable file to the internal storage, the *sink* application reads its contents. Similarly the *External Storage* simply uses a file on the external storage. For the external channel to work, the *source* application requires an extra permission: `WRITE_EXTERNAL_STORAGE`. Again, similar to the *Shared Preferences* communication channel, these channels do not require synchronization between the applications.

Broadcast Intents (Application): the *source* application communicates by adding private data as extra payload to a broadcast message sent to the system. Broadcast intents are a particular type of messages that are used in the Android OS to enable one form of communication between applications. The operating system, upon receiving such a message with its payload, broadcasts it to all the applications that requested to be notified when such a message is received (i.e., by registering themselves for a particular ID that is used to identify the message). The *sink* application registers itself with the system and receives the message sent by the *source*. While both applications need to be running at the same time, no synchronization is required in order for the channel to work.

System Log (Application): the *source* writes a specially-crafted message to the system log that the *sink* then reads to extract the information. The extra `READ_LOGS` permission is required by the *sink* application in order to be able to read the system logs. Messages longer than 4000 characters must be split and binary data must be encoded, because data is otherwise lost when inserted into the log. Given that the log has a finite number of entries that are held at any time, the *sink* application must be activated before the message sent by the *source* is deleted. Alternatively, the *source* could repeatedly insert the message at time intervals to increase the chance that the *sink* receives it. Potentially the channel can be rendered stealthy by filling the log with seemingly meaningful logging data after the communication takes place.

UNIX Socket Communication (OS): the *source* sends the data through a UNIX socket that the *sink* application opened. For this channel to work correctly, both applications must be simultaneously active.

3.2 Covert Channels

We now describe the covert channels that we implemented and measured. As the storage of these channels is not persistent, all these channels are synchronous. This means that before starting to exchange data over the channel a synchronization protocol between the *source* and the *sink* must be run in order to achieve better accuracy during the data-exchange phase. For channels where accuracy is not specifically stated, our implementation reached perfect accuracy.

Single and Multiple Settings (Application): the *source* modifies a general setting on the phone and the *sink* reads it as described in [29]. Multiple settings can be changed at the same time to achieve higher throughput. Most settings in Android can be changed and read without requesting any permissions. This particular covert channel can be closed by disabling or requiring extra permissions in order to change particular settings.

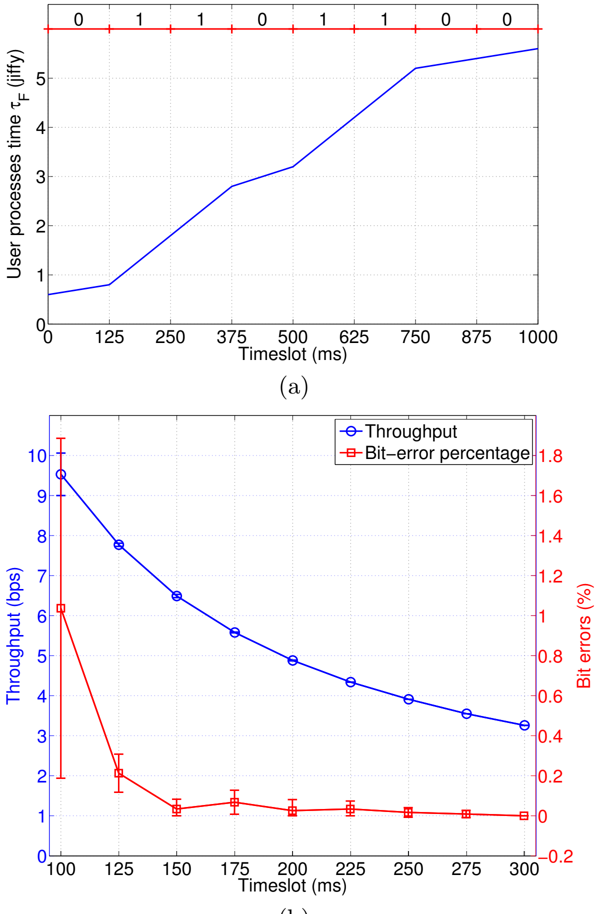
Type of Intents (Application): the *source* sends a broadcast message (similar to the *Broadcast Intents* overt channel) to the *sink* and encodes the data to be transmitted into the type of the intent (i.e., flags, action, particular extra data), rather than directly exchanging the data as the extra payload of the message. In contrast to the similar overt channel that uses *Broadcast Intents*, this covert channel is not detectable by tainting mechanisms or similar solutions. The *sink* application still needs to register with the system in order to receive the intents.

Automatic Intents (Application/OS): the *source* modifies particular settings (i.e., the vibration setting [29]) that trigger automatic broadcasts by the system to all applications that registered to be notified when such a change happens. The *sink* receives the messages and infers the data depending on the contents of the received broadcasts. For instance, changing the vibration setting of the phone triggers a broadcast which contains 1-bit of information (vibration on equals to 1, vibration off equals to 0).

Threads Enumeration (OS): the *source* spawns a number of threads and the *sink* reads how many threads are currently active for the source application by looking into the `/proc` directory of *source*. This particular covert channel can be closed by controlling application access to the `/proc` filesystem or by mediating the access through a system service.

UNIX Socket Discovery (OS): the *source* uses two sockets, a synchronization socket and a communication socket. The *sink* checks if the *source* communication socket's state is open, and infers the transferred bit. The synchronization socket is open if the communication socket can be checked.

Free Space on Filesystem (OS): the *source* application writes or deletes data on the disk to encode the information for the *sink*. The channel throughput depends on the noise in the system; for example the sink application could infer a larger amount of information depending on how many free blocks are available. The data presented in Table 2 was generated by having the *source* allocate three blocks to encode a '1' and clearing three blocks to encode a '0'. The *sink* checked the available blocks in the system at predefined time intervals (75ms for the Nexus One and 100ms for the Galaxy S). The current implementation yields bit-errors percentages between 0.01% (Nexus One) and 0.03% (Samsung Galaxy S). A possible solution for preventing this channel is to enforce a quota on the available space for each application (that could potentially vary depending on the number of applications on the system) and report the free blocks remaining of each quota rather than the free blocks in the overall system.



(a)

(b)

Figure 2: Figure (a) shows an exemplification schematic rise of the value τ_F (number of jiffies spent for every user process) over time when sending the bits written on the top. Figure (b), the graph shows the trade-off between throughput and accuracy (measured in bit errors) for the `/proc/stat` channel. Values are averaged over 5 independent runs.

Reading `/proc/stat` *(OS):* the *source* application performs some computations, while the *sink* monitors the processor usage statistics. These are available in the `/proc/stat` virtual file where the Linux kernel provides information about the current system load (as the number of jiffies used for all user processes). A schematic representation of how the values (τ_F) read in `/proc/stat` change depending on the bit that the *source* wants to send is presented in Figure 2(a). The overall idea is that sending a '1' causes, in the values read, a steeper slope than sending a '0'. Figure 2(b) presents the trade-off between throughput and accuracy of this channel. Other channels behave similarly to this one, with higher throughput resulting into lower accuracy. The current implementation yields bit-error percentages between 0% (Samsung Galaxy S) and 0.10% (Nexus One). Similarly to the *Threads Enumeration* covert channel, this channel could be closed by preventing read access to the `/proc` filesystem.

Covert Channel	**Throughput (bps)**		**Synchronization (ms)**	
	Nexus One	Samsung Galaxy S	Nexus One	Samsung Galaxy S
Type of Intents	3350.85 (± 134.11)	4324.13 (± 555.32)	716.8 (± 168.2)	473.0 (± 249.0)
UNIX Socket Discovery	2610.92 (± 305.25)	1647.78 (± 170.70)	5.2 (± 0.8)	13.9 (± 2.2)
Multiple Settings	239.76 (± 9.41)	284.91 (± 1.90)	314.9 (± 21.8)	302.1 (± 11.0)
Threads Enumeration	157.73 (± 0.97)	139.39 (± 7.40)	71.6 (± 7.1)	110.1 (± 8.8)
Automatic Intents [29]	51.38 (± 0.41)	90.67 (± 0.39)	1083.2 (± 75.1)	435.1 (± 180.8)
Single Settings [29]	46.88 (± 0.31)	65.89 (± 0.73)	267.5 (± 3.2)	273.4 (± 11.9)
Free Space on Filesystem	13.07 (± 0.00)	9.80 (± 0.00)	1038.2 (± 5.1)	1442.7 (± 15.6)
Reading `/proc/stat`	7.82 (± 0.00)	3.26 (± 0.00)	6923.4 (± 8.1)	16669.2 (± 48.7)
Processor Frequency	4.88 (± 0.00)	0.47 (± 0.09)	8203.9 (± 7.2)	78866.1 (± 9156.8)
Timing Channel	3.70 (± 0.00)	3.69 (± 0.01)	10286.8 (± 16.1)	68057.6 (± 105259.4)

Table 2: List of implemented *covert* channels in the Android OS with their corresponding throughput (95% confidence intervals are shown in parenthesis). The displayed values are averaged over 10 runs for both the Nexus One and the Samsung Galaxy S. Channels listed in bold are new channels that we engineered and for which we did not find references in the open literature. In the synchronization column we present the time required to run the synchronization protocol before starting to transmit data through the channel.

Timing Channel (Hardware): the data transmission between the source and the sink is performed by varying the load on the system. The *source* runs CPU-intensive tasks to send the bit ‘1’, on the other hand it does not perform any CPU-intensive operation to send the bit ‘0’. The *sink* continuously runs computation-intensive operations and records the time required to complete them. The *sink* uses this time to infer the presence of computation by the *source* thus inferring the transmitted bit. For reliable differentiation of bits based on the time, an initial *learning period* is used to benchmark the system behavior. Finally, to eliminate the noise in the system, we use a majority vote (out of five measurements) at the *sink* to decide the value of a particular bit depending on a threshold value updated with a moving average. In our implementation, the time difference between transmitting a ‘1’ and a ‘0’ is approximately 6 ms in the case of the Nexus One. The current implementation yields bit-errors percentages between 0.10% (Nexus One) and 0.05% (Samsung Galaxy S).

Processor Frequency (Hardware): this channel is an improvement over the basic *Timing Channel*; in this particular instance we take into account Dynamic Frequency Scaling (used on the smartphones that we tested) to improve the throughput and reduce the synchronization time (for the Nexus One). While the *source* behavior remains the same as in the case of the *Timing Channel*, the *sink* instead monitors the trend of the processor frequency by repeatedly querying it from the system and thereby decodes the current bit. Afterwards the *source* waits a fixed amount of time to allow the CPU to “slow down” again before the next bit transmission is started. The current implementation yields bit error percentages between 0.14% (Nexus One) and 4.67% (Samsung Galaxy S).

3.3 Communication Channel With External Agents

We extend the concept of colluding applications and consider the scenario in which there is only one application installed on the system that has access to private data and wants to disclose it to a third-party web service without requesting the permissions to connect to the network. Furthermore, we want to ensure the successful transmission of the private data through a channel that is hard to detect. Here, the colluding *sink* application resides on a web page executing intensive JavaScript operations, that is opened within the system browser. The phone will show the page on the screen, therefore, to decrease detection by the user, the operation can be carried out when the phone screen is off (for example, during night time). To reach the *sink*, the *source* application uses a covert timing channel similar to the *Processor Frequency* covert channel. However the *sink* cannot directly query the processor frequency, as it is inside the JavaScript sandbox. Such channel is visualized in Figure 1(b).

We have implemented and tested this proposed covert channel as follows: depending on the current bit to transfer, the *source* either tries to increase the processor frequency or sleeps. Afterwards the *sink* measures how many dummy RC4 operations it can perform in a fixed time period, thereby getting the processor frequency and the transmitted bit. The possibility to use the browser to send private data has been described in [19] but their proposed method is easily detected by flow-tracking techniques (such as TaintDroid). Our proposed covert channel—while having a low throughput of roughly 1.29bps on the Nexus One—is also much harder to detect and cannot be detected by today’s tools.

3.4 Results of the Analysis

The experiment results reported in Tables 1 and 2 indicate that the attacker’s choice of one channel over another, depends on the nature and size of data that needs to be transmitted between applications. For example GPS coordinates usually consist of two floating point numbers (represented by 32 bits of data), in contrast contacts, for example, might have a varying number of characters: in order to simulate a few full names and corresponding phone numbers we transmit 135 bytes of information.

Given a rough estimate of the size of the data that can be shared between applications, we conclude that even the covert channels with low throughput, such as the *Timing* or the *Processor Frequency* channels (respectively at around 3.70 and 4.88 bps) enable the sharing of reasonable amounts of data on the smartphone. For example, exchanging GPS coordinates requires roughly 19.4 or 14.8 seconds respectively; sharing 135 byte contacts requires roughly 304.9 or

231.1 seconds respectively. Covert channels with higher throughput, such as the *Type of Intents* or *UNIX Socket Discovery*, reaching up to 4324.13 and 2610.92 bps, enable the exchange of GPS coordinates or contact information in less than a second.

Another interesting result of the analysis is that most channels, when tested on the more powerful Samsung Galaxy S, did not perform better than on the Nexus One. For CPU-bound channels these results come from the fact that Samsung ships the device with a larger number of active services which influence the different channels. For channels based on Processor Frequency it is based on the fact that there is a different frequency governor. For IO-bound channels these results come from the fact that the Samsung device uses a Samsung-developed file system rather then the standard YAFFS2 used on the Nexus One.

Overall, the results show that application collusion attacks, through the usage of different communication channels depending on the amount of data that needs to be transmitted, are a realistic attack and therefore a serious threat.

4. ANALYSIS OF EXISTING TOOLS

In this Section we test two recently proposed tools that try to solve the information leakage on modern smartphones, in particular TaintDroid [10] and XManDroid [4]. We use the Nexus One as the test phone where we successfully install both tools and report the findings.

4.1 TaintDroid

TaintDroid [10] tries to track information flows within an application and between applications; it is implemented as a modification of the Android operating system. Using dynamic taint-tracking, the modified OS follows the information flow of tagged data, that is, data which is generated from sources of private information, including the user contacts and the GPS location.

Inside the Dalvik VM, TaintDroid employs variable tracking and propagates taint through primitive data types, exception handling routines and array lookups. Tainting information, though, does not follow through in native code (such as JNI native libraries) execution. Due to this limitation, at the moment of writing, trying to use native libraries not residing in the `/system` folder results in an application crash. Additionally, taint is propagated through IPC messages, by performing message-level tracking.

Whenever tainted data reaches a sink (such as the network), a notification is shown informing the user about the application that is leaking data, the originating data class and the network transmission. Interestingly enough, the implementation of TaintDroid notifies the user when the *sink* application uses Java's `HttpURLConnection` to send the data off the device, but no detection happens when it uses a UDP connection (i.e., through the Java `DatagramSocket` class). We believe this is just an implementation detail overlooked by the authors rather than a design flaw of the proposed solution.

In our study, TaintDroid was able to correctly report the transmission of sensitive data for the following overt channels: *Internal Storage* and *Broadcast Intents.* The *External Storage* channel was not detected: this happens because taint information is propagated using extended attributes and external storage uses the FAT file-system which does not support them. Surprisingly the remaining overt channels (*Shared Preferences* and *System Log*), which should be detected by TaintDroid were not detected. Further analysis shows that the implementation of the logging mechanism in Android OS is carried out in native code (i.e., C). As previously stated, TaintDroid is not currently capable to extend tagging to native code and therefore cannot detect this channel.

Given that the authors explicitly state that the TaintDroid mechanisms can be circumvented through the use of implicit flows, it is clear that the covert channels implemented in our framework remain undetected due to their bit-wise nature. To remove the taint from tainted variables, such that higher-throughput overt channels can be used successfully, we propose four different techniques. We implement each technique and test it on a Nexus One to report the throughput.

n-way Switch Statement [6]: an n-way Switch Statement can be used to strip the taint off $log(n)$ bits. The Switch Statement reads the tainted value and writes the corresponding constant into a new untainted memory location. The taint does not propagate, because constant values are used. We measured the throughput of this technique to be, roughly, 27.65 Mbps (megabits per second).

Java Exception Handling: Here we encode a tainted bit in the existence of a Java exception. If the tainted bit is '1', an exception is thrown that causes the untainted bit as well to be set to '1' by the exception handler [22]. We measured the throughput of this technique to be, roughly, 107.42 kbps (kilobits per second).

File-based: This technique encodes the tainted bit in the existence of a file inside the application's private directory. Depending on the tainted bit a special file is either created or not. The untainted bit is set depending on the results of the following existence check for the special file. We measured the throughput of this technique to be, roughly, 680 bps (bits per second).

Timing-based: The application's own execution time encodes the tainted bit in this example. The application delays its own execution by sleeping in order to signal a one. Timing measurements determine the value of the untainted bit. We measured the throughput of this technique to be, approximately, 98 bps (bits per second).

Given the throughput of each untainting technique, and the fact that covert channels remain undetected by TaintDroid, we conclude that employing a dynamic flow-tracking technique does not prevent application collusion attacks.

4.2 XManDroid

XManDroid was first presented in [3] and later extended in [4]. It aims at implementing different techniques to successfully mitigate the problems of *confused deputy* attacks and direct *application collusion* attacks.

The authors propose a security framework to enable policy enforcement at different system levels on Android. The instantiation of the framework extends various parts of the Android OS, in particular they port and extend TOMOYO Linux [16]. The security framework modifies Android *reference monitor* to check for direct IPC calls at runtime between applications and indirect communication through Android system components (i.e., the settings manager). Furthermore, kernel-level MAC (Mandatory Access Control) monitors access to different resources such as the file system, UNIX sockets and internet sockets. System policies are expressed in a high-level language and specify which flows are to be denied.

The prototype with which we experimented was able to block a subset of channels that are potentially detected by XManDroid. In particular the prototype successfully detected all the *overt* channels except the *System Log* channel. It also successfully detected the *Type of Intents* and *UNIX Socket Discovery* covert channels, as they work over explicit communication between applications. Further, the *Reading* `/proc/stat` and *Threads Enumeration* covert channels are detected by the fact that they work by accessing the `/proc` file system, blocked by the TOMOYO Linux access control.

Similarly, it is safe to assume that XManDroid would be able to detect the *Broadcast Intents* and *UNIX Sockets Communication* channel, because they work over explicit communication between applications. The *System Log* channel could also be detected because it works over simultaneous access of a shared file (i.e., similarly to the *storage*-based channels that are detected).

This leaves our analysis with a small subset of covert channels that are not detected by XManDroid: *Free Space on Filesystem*, *Processor Frequency* and finally *Timing Channel*. In particular the last two that are *hardware-level* communication channels are not in the scope of XManDroid.

One limitation of using XManDroid and similar tools is that they might report false-positive results when two non-malicious applications try to share legitimate data, as the communication is blocked even if non-sensitive data is shared (i.e., XManDroid is agnostic of the transmitted data). While the authors claim that this is not an issue because non-malicious applications generally do not tend to share data, it might be interesting to understand if it is possible to render XManDroid data-gnostic. Similarly, restricting access through policies to parts of the system (i.e., `/proc`) might result in some applications to malfunction in case they rely extensively on access to such resources. Finally, XManDroid works by specifically adding hooks to system functions or to the kernel as new channels are discovered, and is therefore a reactive solution.

Given that some channels remain undetected under existing state-of-the-art tools, we conclude that application collusion attacks remain a threat and stress that research should focus on closing these (obviously harder to deal with) channels.

5. MITIGATION TECHNIQUES AND THEIR LIMITATIONS

Solving the confinement problem, and in particular closing all possible covert channels in a system, is known to be a difficult problem [9, 20]. It is further complex in the case of smartphones, where performance, application markets openness and exposed API features are key to user and developer adoption. Mitigation can be achieved either at design time (by reducing access to sensitive APIs or by limiting communication possibilities) or by analyzing static and dynamic properties of applications and their interactions off-line or at run-time.

5.1 Design Time Mitigation Techniques

General Purpose Techniques. There are a number of techniques that could be considered by smartphone operating system designers:

User control on private data access: As in Windows Phone 7, involving user action on each data access helps to mitigate the impact of colluding applications (and more generally, malicious applications). However, this also limits applications capabilities; for example, in such an environment, it is impossible for third-party developers to create applications that perform automated backups of private data.

Limiting APIs: When designing APIs exposed to third-party developers, designers should carefully consider the possibility that the API may create a communication channel between applications. If an overt or covert channel is found, it should be either mitigated or its access should be controllable through the system's policies.

Limiting Multitasking: Reduces the possibility of covert channels resulting from competition for access to resources (CPU time, cache and bus contention). However, this limits the diversity of applications that can be implemented on the system.

Application Review: Performed to detect colluding applications before publication of applications on the markets. However, this approach requires dedicated techniques to detect application collusion.

Policy-Based Installation Strategy: Could be used in a corporate scenario where installation of applications can be limited through some policies e.g., deny access to applications that read contacts.

Application-Level Channels. Communication channels constructed at this level are dependent on the APIs exposed by the underlying operating system. Careful design of permissions used to access data sources as well as data sharing points (i.e., sharing of files or preferences, settings, broadcast intents) could draw attention towards applications that require an excessive number of permissions. Furthermore, some of these channels could be closed by removing unnecessary APIs after an analysis of the used and unused ones. This would enable tighter security while maintaining a reasonable amount of freedom for the developers. For instance, the *System Log* channel can be closed by allowing access to the log file only when "USB Debugging" is active. This is a mode to which the phone switches to when connected to a PC, for instance, for development purposes.

Operating-System-Level Channels. Covert channels that can be established at this level might not be detectable by information flow analysis and their prevention requires further investigation. Such channels usually utilize mechanisms offered by the underlying kernel (i.e., sockets, threads, child processes) and therefore, removing such functionality by preventing developers from using it might impede certain applications and their potential optimization. Other system information made available (for example, through the `/proc` filesystem by the Linux kernel) could be restricted (for example, as done in GRSecurity [15], TOMOYO Linux [16] or SEAndroid [31]) or mediated by operating system services that could directly control access to such information.

Hardware-Level Channels. Covert channels (e.g., timing) established at this level are the usually hardest to remove without serious performance degradation or functionality impact. Solutions, such as preventing multitasking or flushing caches between process scheduling, limit the overall performance or responsiveness of the system and increase its power consumption. Furthermore, common data tainting or information flow control techniques are ineffective in this scenario since communication happens at the bit-level of the transmitted data. Closing these channels requires novel approaches, e.g., design of information flow secure systems from the bottom up [33], however redesigning current smartphone systems from scratch is likely to have a prohibitive cost.

One possible solution for timing-based channels (similarly proposed for different systems in [17]) is to add a new permission to the Android OS (for example, it can be named `REQUIRE_PRECISE_TIMING`). Applications requiring such permission, upon requesting timing information, would be given precise timing information (i.e., games require precise timing for physics engines or graphics display). Applications without such permission would be presented with a rough estimate of timing (i.e., ±5 seconds). This modification would disrupt the correct functioning of communication channels that require precise timing for their operation such as, in our reported channels, the *Timing Channel* and the *Processor Frequency* channel. For example, in our implementation the *Timing Channel* works over 6 ms differences (to send a '1' or a '0'), therefore if the system would report time to applications at a granularity higher than 6 ms the channel would be disrupted.

If further analysis shows that precise timing is indeed required for the correct functioning of a large number of applications, another viable solution to disrupt timing channels is to limit the number of times applications can request timing information.

5.2 Application Analysis Techniques

Black-box Analysis. One strategy in trying to detect collusion is to add a data monitor between separate applications on the device. This would remove the need to detect covert channels by only monitoring data leakage itself. In such an architecture, when data from one application is used, the monitor would store it (or a fingerprint of it) and track the data sent to the colluding application. While this approach seems promising, it is inherently limited: malware can encode data in a way that it leaves the mobile device still encoded (e.g., encryption using a public key), defeating the monitoring. While it is clear that black-box analysis may detect some trivial attempts to evade the system security policy, it clearly does not provide a complete solution.

Exclusive access to sensitive resources. Techniques to limit access to sensitive resources (e.g., the microphone) from third-party applications when a sensitive operation is ongoing (e.g., a phone call), as presented in [29], only prevents malware from accessing that particular data at that instant. Such techniques cannot be applied generally: for example, access to the GPS data would always be considered a privacy invasive operation and therefore would never be allowed.

Offline application analysis. Since colluding applications are communicating on an unexpected channel, it is likely that when colluding applications are executed simultaneously on a device, they would show a different behavior than when executed independently. For example, they would detect each others presence and engage into communication over a covert channel. Behavioral analysis could be used to detect such a change of behavior, for example executing applications on an emulator alone or in pair and comparing execution traces and coverage. However, given the vast number of potential pairs of colluding applications, this solution does not scale. This can be addressed by strategies which include evaluating applications according to their popularity or according to "replicated" installations [28].

6. RELATED WORK

The Confinement Problem and Covert Channels. Lampson first described the confinement problem [18] as the problem of preventing unauthorized communication, over overt or covert channels, between two subjects on a system. It is recognized to be a difficult problem in practice, Denning and Denning state that "Cost-effective methods of closing all covert channels completely probably do not exist" [9].

While overt channels can be managed by security policies, *covert channels* are communication channels built from resources that are not intended for communication, and so they cannot be mitigated with the same techniques. Covert channels were also used to perform covert communications over networks [27, 12], however in this work we mainly focused on inter-process covert channels. Inter-process covert channels can be classified as either software (sometimes referred to as TCB channels) or hardware (also known as fundamental channels) and communicate over timing or storage channels. However, this distinction is more empirical than theoretical [13].

Software covert channels can be mitigated by a careful analysis of the usage of visible and alterable variables used by system calls [34] or using a formal model for analyzing programs [30] using a semi-automated technique. However, hardware-related covert channels (e.g., timing, competition to access resources, paging) are difficult to prevent and recent processor designs have been shown to increase the number and efficiency of covert channels [36].

As an example, multi-core application processors are already available for smartphone devices, which would render covert channels over cache highly reliable [36]. Possible mitigation techniques include using fuzzy time [17] and preventing multitasking.

Permission-Based Security. A significant amount of work has been performed, in the past few years, on the Android platform and specifically on the permission-based model [2, 5, 7, 8, 24, 26, 35]. Barrera et al. present an empirical methodology for the analysis and visualization of the permission-based model, which can help in refining the permission system [2]. The *Kirin* tool [11] uses predefined security rule templates to match dangerous combinations of permissions requested by applications. As *Kirin* analyzes individual applications, colluding applications would not be detected by such policies. Saint [26] allows run-time control over communication among applications according to their permissions. In [5], Burns discusses possible unchecked information flows due to applications that use *Broadcast Intents* without proper permissions checking.

Soundcomber[29] introduces a proof of concept malware based on application collusion for Android smartphones. The authors foresee using the microphone of the device to harvest sensitive information, such as credit card numbers, by detecting voice and tone patterns. The information is then sent over an application with the necessary permissions to send it through the internet through means of covert communication channels. The channels presented by the authors use globally-available settings (vibration, volume, screen lock, etc.) or file locks. In contrast, we presented a wide range of channels and carried out an analysis of their behavior in terms of throughput and accuracy through our framework.

7. CONCLUSION

We demonstrated that application collusion attacks against the permission-based mechanisms used on modern operating systems for mobile devices, such as Android OS, are a serious threat given different implementation of communication channels at different system levels. The results of the throughput measurements for each channel show that even covert channels with low throughput are still sufficient to exchange possibly private information stored on a smartphone. Finally, we confirmed that proposed and implemented techniques and tools do not provide a complete solution against different communication channels and are therefore insufficient to prevent application collusion attacks, which remain an open problem for the research community.

Acknowledgments

This work was partially supported by the Zurich Information Security Center (ZISC). It represents the views of the authors. We would like to thank Sven Bugiel and the team behind XManDroid for giving us the possibility to test XManDroid and for productive discussion.

8. REFERENCES

[1] J. Anderson, J. Bonneau, and F. Stajano. Inglorious Installers: Security in the Application Marketplace. In *Workshop on the Economics of Information Security*, WEIS '10, 2010.

[2] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to Android. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 73–84, New York, NY, USA, 2010. ACM.

[3] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. XManDroid: A new Android evolution to mitigate privilege escalation attacks. Technical Report TR-2011-04, Technische Universität Darmstadt, April 2011.

[4] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on Android. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, NDSS '12, February 2012.

[5] J. Burns. Developing secure mobile applications for Android. `https://www.isecpartners.com/files/iSEC_Securing_Android_Apps.pdf` (accessed October 2012), 2008.

[6] L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pages 143–163, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] A. Chaudhuri. Language-based security on Android. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 1–7, New York, NY, USA, 2009. ACM.

[8] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on Android. In *Proceedings of the 13th International Conference on Information Security*, ISC'10, pages 346–360, Berlin, Heidelberg, 2011. Springer-Verlag.

[9] D. E. Denning and P. J. Denning. Data security. *ACM Comput. Surv.*, 11(3):227–249, Sept. 1979.

[10] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[11] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 235–245, New York, NY, USA, 2009. ACM.

[12] C. G. Girling. Covert Channels in LAN's. *IEEE Transactions on Software Engineering*, 13(2):292–296, 1987.

[13] V. Gligor. A guide to understanding covert channel analysis of trusted systems, version 1 (light pink book). NCSC-TG-030, Library No. S-240,572, November 1993. National Computer Security Center, TCSEC Rainbow Series Library.

[14] Google. Android OS (up to version 2.3.7). `http://developer.android.com/`.

[15] GRSecurity. The GRSecurity project. `http://grsecurity.net/features.php`.

[16] T. Harada, T. Horie, and K. Tanaka. Task oriented management obviates your onus on linux (TOMOYO Linux). Linux Conference, 2004.

[17] W.-M. Hu. Reducing timing channels with fuzzy time. In *Proceedings of IEEE Computer Society Symposium on Research in Security and Privacy*, pages 8–20, May 1991.

[18] B. W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, Oct. 1973.

[19] A. M. Lineberry. These aren't the permissions you're looking for. BlackHat USA, August 2010.

[20] S. B. Lipner. A comment on the confinement problem. In *Proceedings of the 5th ACM Symposium on Operating Systems Principles*, SOSP '75, pages 192–196, New York, NY, USA, 1975. ACM.

[21] Microsoft. Security for Windows Phone 7. `http://msdn.microsoft.com/en-us/library/ff402533%28v=VS.92%29.aspx` (accessed October 2012).

[22] S. K. Nair, P. N. D. Simpson, B. Crispo, and A. S. Tanenbaum. A virtual machine based information flow control system for policy enforcement. *Electron. Notes Theor. Comput. Sci.*, 197(1):3–16, Feb. 2008.

[23] Nokia. Symbian OS. `http://symbian.nokia.com`.

[24] J. Oberheide. Android Hax. SummerCon 2010, June 2010. `http://jon.oberheide.org/files/summercon10-androidhax-jonoberheide.pdf` (accessed October 2012).

[25] J. Oberheide and F. Jahanian. When mobile is harder than fixed (and vice versa): demystifying security challenges in mobile environments. In *Proceedings of*

the 11th Workshop on Mobile Computing Systems and Applications, HotMobile '10, pages 43–48, New York, NY, USA, 2010. ACM.

[26] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in Android. In *Proceedings of the 25th Annual Computer Security Applications Conference*, ACSAC '09, pages 340 –349, dec. 2009.

[27] F. A. Petitcolas, R. J. Anderson, and M. G. Kuhn. Information hiding-a survey. *Proceedings of the IEEE*, 87(7):1062 –1078, July 1999.

[28] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid Android: versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 347–356, New York, NY, USA, 2010. ACM.

[29] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, NDSS '11, pages 17–33, Feb. 2011.

[30] A. B. Shaffer, M. Auguston, C. E. Irvine, and T. E. Levin. A security domain model to assess software for exploitable covert channels. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '08, pages 45–56, New York, NY, USA, 2008. ACM.

[31] S. Smalley, NSA, and Trust Mechanisms (R2X). SEAndroid. `http://selinuxproject.org/page/SEAndroid` (accessed October 2012).

[32] The Lookout Blog. Lookout's privacy advisor protects your private information. `http://blog.mylookout.com/2010/11/lookout%E2%80%99s-privacy-advisor-protects-your-private-information/` (accessed October 2012).

[33] M. Tiwari, J. K. Oberg, X. Li, J. Valamehr, T. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood. Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 189–200, New York, NY, USA, 2011. ACM.

[34] C.-R. Tsai, V. D. Gligor, and C. S. Shandersekaran. On the identification of covert storage channels in secure systems. *IEEE Transactions on Software Engineering*, 16:569–580, June 1990.

[35] T. Vennon and D. Stroop. Threat analysis of the Android market. Technical report, GTC, June 2010. Smobile systems technical report, Available at `http://threatcenter.smobilesystems.com/wp-content/uploads/2010/06/Android-Market-Threat-Analysis-6-22-10-v1.pdf` (accessed October 2012).

[36] Z. Wang and R. B. Lee. Covert and side channels due to processor architecture. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, ACSAC '06, pages 473 –482, dec. 2006.