Hiding Privacy Leaks in Android Applications Using Low-Attention Raising Covert Channels

Jean-Francois Lalande
ENSI de Bourges, Univ. Orléans, LIFO, EA 4022
F-18020, Bourges, France
jean-francois.lalande@ensi-bourges.fr

Steffen Wendzel
Fraunhofer FKIE / Cyber Defense Group
Friedrich-Ebert-Allee 144, 53113 Bonn, Germany
steffen@wendzel.de

***Abstract*—Covert channels enable a policy-breaking communication not foreseen by a system's design. Recently, covert channels in Android were presented and it was shown that these channels can be used by malware to leak confidential information (e.g., contacts) between applications and to the Internet. Performance aspects as well as means to counter these covert channels were evaluated. In this paper, we present novel covert channel techniques linked to a minimized footprint to achieve a high covertness. Therefore, we developed a malware that slowly leaks collected private information and sends it synchronously based on four covert channel techniques. We show that some of our covert channels do not require any extra permission and escape well know detection techniques like TaintDroid. Experimental results confirm that the obtained throughput is correlated to the user interaction and show that these new covert channels have a low energy consumption – both aspects contribute to the stealthiness of the channels. Finally, we discuss concepts for novel means capable to counter our covert channels and we also discuss the adaption of network covert channel features to Android-based covert channels.**

***Keywords*-Smartphone Security; Android; Covert Channels; Privacy; Information Hiding**

I. Introduction

With the increasing number of Android smartphones, the threat for user's privacy is growing. Users face the problem of spyware aiming to steal sensitive data or malware that tries to use the available functionalities of the phone. For example, the malware *Walkinwat* [1] sends short messages to each contact of the user's address book. These short messages inform the contacts that the user owns a pirated version of *Walk and Text*. In [2], Morrow reports that 64% of the enterprises surveyed by Infonetics had data lost or stolen because of the use of mobile devices. After data got leaked, it is out of the user's control and an attacker can, for instance, sell the leaked data, can use the data to blackmail the user, or can use the data to get business advantages.

To prevent the operation of malware, the user is responsible of reviewing the application's permissions at the time of installation. Nevertheless, the majority of the users will probably skip such review steps since they require technical knowledge [3], [4]. The user may also think that, for update purposes, the application requires Internet access [5].

Thus, a lot of papers deal with the detection of malware or the detection of vulnerabilities in Android. The proposed solutions are of different natures and can be applied for automatically detecting privacy leaks. The mechanisms can be classified in three groups. They can: statically analyze the application [6], [7], [8], [9], [10], for example, to detect potential privacy leaks; dynamically analyze the application [7], [8], [10], for instance, to prevent privacy leaks; modify the operating system [11], [12], [13], for instance, to improve the enforcement of more fine grained security policies.

Applying one of these solutions to the problem of privacy leaks consists in detecting (preventing) that a flow of information occurs between *private* data and *public* containers, e.g., between contact information and the Internet. One of the most popular solutions is *TaintDroid* [11]. TaintDroid applies realtime checks to detect whether private data is leaked by an application. If TaintDroid notices that a flow of information occurs between a resource labeled as private and a resource labeled as public, the flow is prevented. Another similar contribution is called *CHEX* [9]. CHEX enables the tracking of vulnerabilities using static analysis in components of applications that may leak private data. Even if these approaches provide a powerful means for detecting a malware that violates the user's privacy, this paper aims at discussing the possibility of defeating control flow monitoring solutions.

Therefore, we present the design and implementation of a malware capable of disclosing private information while bypassing detection solutions such as TaintDroid [11] and CHEX [9]. As done in [14], our proposal is to split the malware into *two parts* and to establish a covert channel between the two applications. The first part of the malware is responsible of collecting the private data and transfers it to the second application that discloses the private data to the Internet. As an improvement over [14], we propose to design a covert channel with minimal permissions that is correlated to user interactions in order to be stealthy. Additionally, our presented covert channels comprise separated data and control channels to achieve a high quality of transmission. We also show that our application raises only few attention, has a low energy footprint (what increases the covertness) and that it is thus difficult to detect.

Our scenario of using two applications is realistic as one application can include advertisements for the other application (e.g., to enhance features) and since a shared website can offer both applications. For example, a website can present a set of energy saving applications which

978-0-7695-5008-4/13 $26.00 © 2013 IEEE
DOI 10.1109/ARES.2013.92

are capable to cooperate; one application could visualize the energy consumption of the user's smart home while another application could be an energy advisor or a remote control for the heating, ventilation, and air-conditioning (HVAC) system of the user.

The remainder of this paper is structured as follows. Section II is split into three parts and introduces related work for Android security, covert channels, and covert channels *in* Android. Section III discusses the scenario in which the covert channel's and the data leakage are established as well as the properties of the malware. Section IV describes the malware architecture and the implemented covert channel techniques. We cover additional experiments done to evaluate the presented covert channels in Section V and means to counter the covert channels as well as possible means to improve the capabilities of our channels in Section VI. Section VII concludes.

II. Related Work

The research on Android security and on covert channels is based on their particular roots which will be discussed separately. Afterwards, we survey related work that combines covert channel research with Android research.

A. Android Security

1) Static and dynamic analysis: In order to build innovative protection solutions for the Android platfom, the existing research studies the possible vectors of attack using static or dynamic analysis techniques.

Papers that are based on static analysis techniques try to build automatic reports about applications. It may help the user to better understand what applications are doing and may also help to discover malware. In [7], basic analysis techniques such as calls to JNI components or *System.getRuntime().exec()* calls help to build vectors that describe the application. Collecting static information is usefull for building visual reports as shown by [6] for the example of an application performing a privacy violation.

Static analysis techniques can be combined with dynamic analysis (as proposed in [7] – one of the first papers dealing with Android and dynamic monitoring, that shows the counting of system calls for a *fork bomb* malware [15]). Recently, [10] proposed *DroidMat* that combines static and API call analysis in order to automatically classify applications and malware in clusters, achieving better results than another approach called *Androguard*.

When a vulnerability or a suspicious behaviour is noticed, the proposed prevention systems try to detect or prevent attacks. For example, Apex [12] enables to define contextualized constraints on the permission granted to applications (localization, number of resource used). To achieve this, the operating system should be adapted to include the countermeasure.

2) Data leakage: As stated in the introduction, one of the most important threats for Android users is the possible leak of private data from their smartphone. Some recent contributions are focusing on workflow monitoring of personal data. One of the most popular solutions is TaintDroid [11] and its derivatives like YAASE [13] that allows to define improved security policies. These solutions track private data leakage by tainting the resources and propagating these taints into the program and in the API offered by the operating system. In [8], an extension to TaintDroid is presented, which includes support for indirect flows that leak personal data by sending information from one container to another using the control flow instructions. In this paper, we show that workflow monitoring solutions are not 100% accurate and can be bypassed by attackers and also show that solutions like TaintDroid can be defeated by our proposed malware.

The data that is targeted by attackers can be personal information like contacts, SMS or mobile phone information like the IMEI. Mulliner studied the data leakage of mobile phones using WAP proxies and discovered that such proxies were configured to leak mobile phone numbers (MSISDN) and subscriber numbers to website operators [16]. Such information can be used for longterm user tracking and can even support lookups for the identity of website visitors.

A lot of regular applications of the *Play Store* include data leakage features. Data leakage of Android applications was evaluated by Stirparo and Kounelis in [17]. They analyzed the top 50 Android applications of different categories (e.g., business or shopping) regarding to their potential to leak data *at rest*, i.e., data that is not currently in use or in transit but saved on memory and can thus be analyzed using forensic tools. In this case, an adversary can try to steal account data when a mobile phone got lost. Stirparo and Kounelis used basic open source tools and discovered essential data leakage in major applications, such as Twitter or Dropbox, which can lead to identity theft [17].

B. Covert Channels

Covert channels are channels not foreseen to be used for a communication [18] but for other purposes like to transfer control information. The importance of covert channels grew within the last years since these channels support multiple scenarios: For instance, covert channels support the freedom of speech since they allow to bypass Internet censorship [19] but also enable botnets to implement a stealthy command and control communication [20].

Covert channel techniques were discussed for decades and were found in many areas of computer systems, such as in local systems [21], in IPv4 and TCP [22], in IPv6 [23] and in business processes [24] – just to mention a few. Means to detect, limit, and prevent covert channels were also discussed for decades and comprise methodology that can be applied at different steps of the software development lifecycle, such as the Shared Resource Matrix Methodology (SRM) [25], Covert Flow Trees [26], the Pump [27], and the application of machine learning and statistics for timing channel detection [28], [29].

Up to now, only few covert channels and anti-covert channel means have been studied in Android systems. The next section will cover Android-specific covert channels.

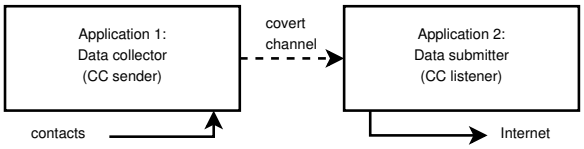

Figure 1. Malware architecture

C. Covert Channels and Android

The first paper that presented a malware based on a local covert channel in Android describes Soundcomber, a malware that captures entered digit numbers using the sound of the smartphone. The paper discusses different covert channels based on the utilization of applications (e.g., a browser) and system functions like the vibration settings, the volume settings, or the screen state [30]. We also use the screen state to implement a covert channel but improve the stealthiness of the channel.

A more recent paper written by Marforio et al. [14] is the first to exhaustively present the possibilities of using local covert channels in Android smartphones. They show how two applications can exchange data using overt or covert channels in order to leak private data. This work is very close to our own work because our malware design, presented in Section IV, is based on the same mechanism: a local covert channel between an application that collects data and another one that discloses the data. Such an architecture is visualized in Figure 1 where the contacts are collected by application 1, are afterwards sent to application 2 using a covert channel, and are finally disclosed on the Internet.

Table I presents a synthesis of the proposed overt and covert channels of [14]. The first six channels could be monitored using a taint propagation solution like TaintDroid [11]: it requires to modify the taint propagation mechanism in order to include the way the covert channel operates. For example, for broadcast intents, it would be necessary to taint each intent. Using such detection mechanism could lead to numerous false positive alarms. The last four proposed channels exploit the reading capabilities of the state of the operating system. Thus, it becomes very difficult to taint such resources. Nevertheless, Marforio et al. propose simple countermeasures to restrict the access to operating system information.

The proposed covert channel of [14] can be defeated by using a taint tracking solution or by restricting the access to special device files in */dev* that provide information about the operating system. In this paper, we show how to build a covert channel that overcomes these limitations. Our goal is to propose covert channels that 1) cannot be easily defeated by restricting access to the filesystem or by solutions such as TaintDroid; 2) have a low footprint with, as a consequence, a low throughput; 3) are difficult to detect by the user. The next section presents our considered use case and the properties for our malware.

III. Considered scenario and malware properties

Before moving to the description of the proposed malware architecture in Section IV, this section describes the properties that a malware should meet in order to be most efficient. Its efficiency can be appreciated by measuring the malware's capacity to be invisible from the user's point of view. Thus, we do not consider the throughput of the covert channel as the primary goal. The efficiency of the throughput was already discussed in [14] and the results are recalled in Table I.

In our scenario, which is the same assumption as in [14], we suppose that a user has installed two applications that embed malicous code. In order to make the malware active, the attacker should include the malicious codes in applications that have great chances to be installed on the same system. For example, the user may installed several games of the same type in order to compare them. Moreover, the attacker can split its application in several applications where each additional application can be seen as a sort of plugin or extension for the primary one. A thinkable example scenario is a website comprising a set of applications that visualize, monitor or improve the user's energy consumption in his smart home (e.g., application 1 could be an energy advisor while application 2 could enable the remote control of the heating).

When performing the installation, a smartphone user will have the opportunity to review the requested permissions of the first application that will later act as the covert channel sender. If the requested permissions are obviously inadequate, he may suspect the inclusion of a malware hidden in a regular application.

Experienced users may investigate the resource consumption of applications. Using too much bandwidth or consuming too much energy may reveal that a malware is operating [31]. This can happen for the two last covert channels reported in Table I. Finally, the malware should not disturb operations of the Android sytem and of other applications. Thus, we can compile requirements for the malicious applications:

- minimize the required security permissions;
- minimize the battery consumption and network load;
- minimize the disruptions of the smartphone use;
- maximize the throughput of privacy leaks.

To achieve these goals, we discuss how such a malware can be designed in Section IV. Section V presents the implementation details and experimental results for our malware. The last section discusses anti covert channel techniques and the possible means to detect our malware.

IV. Malware Design

After the previous section defined the scenario in which our malware operates and the properties linked to the scenario, we will now explain the architectural view of the malware and how our proposed covert channels are realized. Therefore, four different covert communication means are presented. Finally, we discuss the hiding aspects of the different covert channels.

Table I
PROPOSED OVERT AND COVERT CHANNELS IN [14]

Overt/covert channel type	Required permission	Side-effect	Can be tainted	Possible countermeasure	Throughput
Shared settings			Yes		> 100 bps
System log	READ_LOGS		Yes		
File	WRITE_EXTERNAL_STORAGE		Yes	TaintDroid [11]	
Broadcast intents			Yes		10-100 bps
Event intents		May raise attention	Yes		
UNIX socket			Yes		> 100 bps
Thread enumeration			No	Restrict access to /proc	> 100 bps
Free space			No	Apply quota	10-100 bps
Processor statistics		Impacts battery	No	Restrict access to /proc	< 10 bps
Processor frequency		Impacts battery	No	Restrict access to /sys	< 10 bps

A. Malware Architecture

Figure 1 shows the basic workflow of information of our malware. The first part of the malware is an application responsible for the collection of private data, e.g., the contact information. Thus, this application requires the READ_CONTACTS permission. This first part of the malware is also the covert channel sender as it will send the collected data via a covert channel to the receiver.

The second part of the malware is responsible for receiving the hidden data using the covert channel and is called the covert channel receiver. It is also responsible of disclosing the data to a remote server. Thus, this application needs the INTERNET permission. The proposed architecture enables a malware that will be able to create a privacy leak.

Depending of the nature of the covert channels, some additional permissions may be necessary in one or both parts of the malware. Of course, as we try to meet the first property described in Section III, our goal is to avoid or to minimize the use of any permission that would make the user suspect that a malware is present.

B. Covert Channel Setup

If application 1 is allowed to access a confidential information (e.g., the address book), the access permissions are enforced by the Android system. Thus, using a direct communication between application 1 and application 2 will give application 2 access to confidential information using delegation. If application 2 has access to the Internet, the delegation of access can help the malware to create privacy leaks.

In the following, the covert channel is responsible for the transmission of a message m, for example, a contact name, from the CC sender to the CC receiver. For each character x of m, we describe the way the covert channel operates.

We additionally split our covert channel in a control channel used to signal whether covert information is currently transmitted, or not, and a data channel that transfers the actual hidden information. This is similar to existing network covert channel techniques which comprise an internal control header (the control channel) and a payload area (the data channel) like presented in [32], [33]. Table II summarizes the control and data channels of the covert channels discussed in this Section.

Table II
CONTROL AND DATA CHANNELS OF OUR COVERT CHANNELS

Covert channel type	Control channel	Data channel	Required permission
CC#1: Task list/screen	screen state	task list	GET_TASK
CC#2: Process prio./screen	screen state	process prio.	
CC#3: Process priorities		process prio.	
CC#4: Pure screen-based		screen based	WAKE_LOCK

1) **CC#1** - *Task list and screen-based covert channel:* This section describes the design of a covert timing channel exploiting the screen state (*on* or *off*) in combination with the ability to examine the running tasks of the operating system. Examining the tasks of the system requires to add the permission GET_TASK to the CC receiver. Nevertheless, this new permission does not reveal directly that the CC receiver is a covert channel listener. Such a permission is often used by administration tools, for example by task killers or backup tools.

The covert channel is established between the two applications when the screen goes off and the CC sender is in the foreground. The CC sender starts to count the elapsed time and checks whether the screen stays switched off. It waits $x * \Delta T$ ms where x is an integer representing the encoded information that is transmitted and ΔT is a constant that both applications know. When $x * \Delta T$ ms is elapsed, the CC sender kills itself. The CC receiver also monitors the elapsed time from the point where the screen went off. Using the GET_TASK permission, the CC receiver can determine that the CC sender is still running. When the CC sender disappears, the CC receiver can deduce x by dividing the counted time by ΔT.

If the user interrupts the CC sender by switching on the screen, both applications cancel their time measurements. Indeed, if the CC sender is killed when the screen is on, the CC receiver cannot distinguish whether $x * \Delta T$ ms have been elapsed or if the user decided to terminate the application. Thus, both applications consider that the information is not reliable and the CC sender must repeat the transmission of the x value.

Figure 2 shows a possible scenario for screen and CC sender states. At the beginning, the user launches the CC sender (screen is already on). Then, the screen is switched off but is resumed quickly. Thus, the CC sender has not enough time to kill its process. Afterwards the screen goes

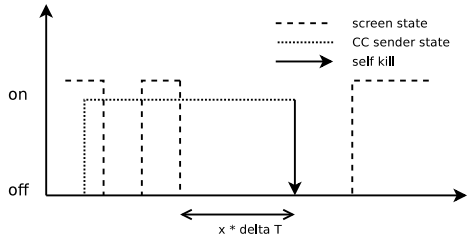

Figure 2. Example of screen and application states

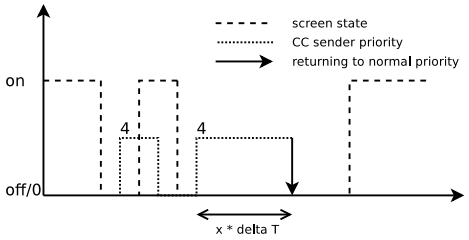

Figure 3. Example of screen and process priority states

off again. After $x * \Delta T$, the CC sender kills itself. Later, the screen goes on.

Since the scheduler is not easily predictable, covert channels that use such synchronization cannot guarantee a 100% accuracy. Nevertheless, using a reasonable value for ΔT and synchronizing the channel with the screen guarantee to operate with a low workload. Thus, the proposed covert channel works flawlessly. However, if the workload increases and ΔT is small, problems will arise on a real-world system.

This covert channel has two specificities that may help the user to discover it. First, it requires the permission GET_TASK for the receiver which is stated when installing the CC receiver. Second, the CC sender kills itself after a certain amount of time if the user switches off or locks the smartphone. This behaviour is not so suspicious for an Android smartphone because the operating system is allowed to destroy any activity in order to recover some extra memory. These aspects are discussed later in Section IV-C where we present how our malware can be hidden in an efficient manner.

2) **CC#2** - *Process priority and screen-based covert channel:* We propose another variant of a covert timing channel that does not require any extra permission, i.e., that eliminates the need for the GET_TASK permission. As for the previous covert channel, we propose to synchronize the two applications using the extinction of the screen state. Then, the CC sender changes its UNIX priority to a value p known by both, the CC sender and the CC receiver, by calling the Process.setThreadPriority(p) static method. This way, the CC receiver can iterate over all possible UNIX process IDs in order to detect the appearance of process priority p by calling the Process.getThreadPriority(uid) static method.

Figure 3 shows an example where the CC sender changes its priority to 4. If the screen goes on too early, the sending of the message is canceled. When the priority is set to 4 during a certain amount of time and returns to 0, the CC receiver can deduce x by measuring $x * \Delta T$.

Since it is thinkable that a regular process is also running with the priority p, the covert channel should be configured to choose an uncommon priority value for p, i.e., a value used by as few processes as possible. The drawback is that it cannot ensure that no other process exists that uses priority p as well. Therefore, parity codes could be applied to detect error-prone data transmitted.

3) **CC#3** - *Process priority-based covert channel:* The third technique we propose is a pure process priority-based covert channel. Compared to CC#2, we do not use the control channel to start the transmission. Thus the CC receiver will monitor the priorities of all possible UNIX processes in order to detect the appearance of the known value p. Then, the CC receiver measures $x * \Delta T$ in order to deduce x.

As discussed in section IV-C, this covert channel has a high throughput but may raise attention because of its energy consumption.

4) **CC#4** - *Pure screen state covert channel:* CC#3 was a derivative solution of CC#2 where we removed the use of the screen state. The last covert channel we present, CC#4, is a derivative of CC#2 where we keep only the screen state functionality. The CC sender waits for the screen to turn off during a given amount of time meaning that the user is probably away. Then, it switches on the screen, waits that it returns off and counts the $x * \Delta T$ delay. Then, it switches on the screen again to inform the CC receiver about the end of the transmission. Thus, the CC receiver just needs to monitor the state of the screen to receive x.

This channel may seem to be the most efficient. Nevertheless, waking up the screen requires the WAKE_LOCK permission (for regular applications) or the DEVICE_POWER permission (for system applications). Moreover, the user may discover the channel by noticing suspicious actions of the screen.

C. Malware Hiding Capabilities

This section discusses the characteristics of the proposed malware that addresses the properties presented in Section III. It also presents how to increase the stealthiness of the malware.

1) Permission: Using the GET_TASK permission for CC#1 can help to exploit covert channels that are linked to the operating system capabilities. Other types of permissions relative to smartphone sensors can also be exploited, for example using the smartphone camera. Nevertheless, using such extra permissions could reveal the presence of malicious code. Thus, we show with CC#2, CC#3, and CC#4 that Android offers primitives that enable to easily create a covert channel that does not require extra permission.

2) Battery consumption: Our proposal has a very low impact on the battery usage as the covert channel has a low throughput: the covert channel is not based on CPU usage and runs limited code each ΔT time while its battery consumption is small if the choosen ΔT is high. Section V-B presents results regarding the energy consumption.

Moreover, the synchronization of CC#1 and CC#2 is based on the screen usage. Thus, the malware is only consuming energy after the user interacts with the smartphone and will stop consuming energy when a byte has been received. The consumption stays limited and correlated with the user actions. Correlating the malware consumption to the user interaction helps to increase the invisibility. On the contrary, CC#3 and CC#4 do not use the control channel that is correlated to the user interaction. These two covert channels will increase their throughput and energy consumption which may raise the attention of the user.

3) Hiding the malware: Using the covert channel CC#1, the attacker should not use a small ΔT. As the CC sender will kill itself, the user may suspect an issue when switching back the screen on. Using a large value for ΔT (more than a minute) could help the user to forget the state of the application he left in foreground.

CC#2 provides the best stealthiness from a user's perspective: It only operates when the screen goes off, receives a part of the message and stays idle until the next user interaction takes place. Even if it consumes more energy than CC#1 (cf. Section V-B), it is the covert channel with the highest stealthiness of the four.

Moreover, CC#1 and CC#2 will stop any operation if the screen goes back on. Thus, the smartphone is not impacted when the users is using it.

V. Experiments

In this section, we first cover implementational details on which our analyses are based. Afterwards, we discuss results regarding the throughput of the channels, their energy consumption and the channel's hiding potential in the context of TaintDroid.

A. Implementation Details

All presented covert channels have been implemented and tested by using the *Android Emulator* as well as by using a regular Samsung Galaxy SIII. Figure 4 gives the details about the algorithms of CC#1 and CC#2[1]. At each step of the algorithm, the CC receiver checks the condition four times more frequently than the CC sender, in order to improve its accuracy in calculating $x * \Delta T$.

To be persistent, both sender and receiver are implemented using services. When the screen is on and is unlocked, the services are stopped. When the screen goes off or if the user terminates one of the applications, the onPause() method is called and the services are launched. For CC#3 and CC#4 the services are persistent. CC#3 can work even if the user interacts with the smartphone whereas CC#4 waits for a period of inactivity.

[1]A video of CC#1 / CC#2 can be viewed at url1: http://dai.ly/x10lbre. A QR code for this url can be found at the end of this paper.

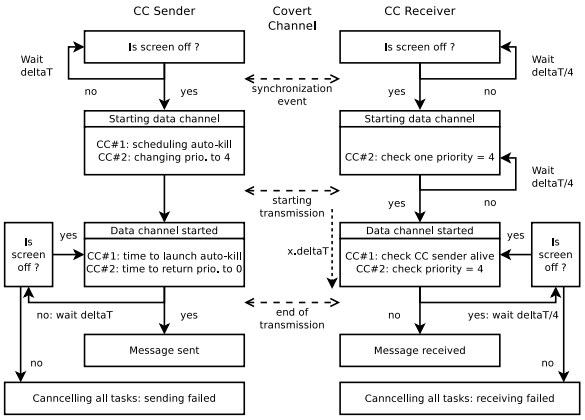

Figure 4. Implemented algorithms for CC#1 and CC#2

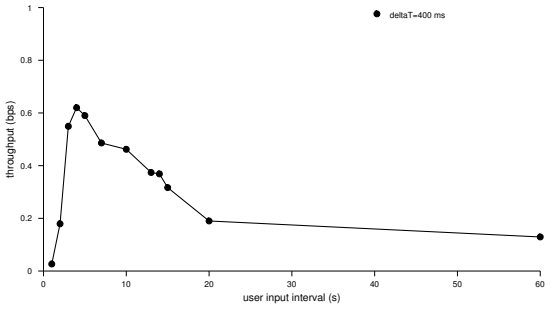

Figure 5. Throughput of the process-based covert channel CC#2 during a period of 3h20min of experiment

B. Statistics

1) Throughput and user interaction: As the covert channels CC#1 and CC#2 rely on the user interaction, the throughput is correlated to the use of the smartphone. To evaluate their throughput, we simulated the interaction of the user that switches on/off the screen.

In Figure 5 we show the throughput of CC#2 in bits per second during an experiment of 3 hours 20 minutes. At regular intervals, a fake user is switching off the screen and waits during a certain amount of time ΔU. Then, the fake user wakes up the screen during two seconds and begins again the process. Using this fake user, we measured the throughput of an infinite alphanumeric random message transmitted using different values of ΔU.

If the value of ΔU is too small, it impacts the throughput because the user breaks the transmission by waking up the screen. The results show that, for $\Delta T = 400\,ms$, the best throughput is obtained for a user interaction interval ΔU of $4\,s$. For a large ΔU, the throughput is low but has better energy characteristics, as described later in V-B2.

However, we used an equally distributed set of symbols to transfer our hidden message. If we reduce the number of symbols we can transfer (e.g., no numbers or only upper case letters), we can improve the bandwidth of the covert channel since we are not required to distinguish

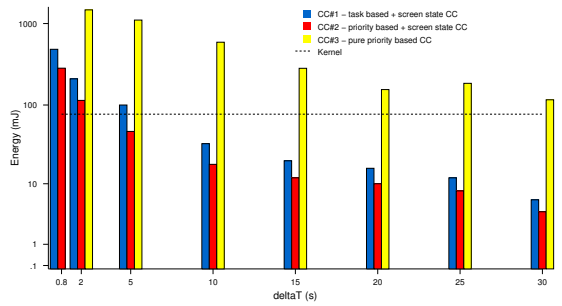

Figure 6. Energy consumption during a 1 minute period of transmission

as many timing delta values as in the current version. Besides, since some symbols are usually more likely to occur than other symbols (e.g., in the English language, the letter 'e' is more likely to occur than the letter 'z' [34]), an improved coding (such as Huffman) would additionally improve the channel's bandwidth. Besides, CC#2 can also use the whole interval of possible priorities for a process (-20 to +19) in order to better encode the information.

For CC#3, using $\Delta T = 800$ ms makes the application running smoothly using a Galaxy SIII. For the Galaxy SIII, we measured a throughput of 17.6 bps which is greater than the throughput of CC#1 and CC#2. Increasing the throughput of CC#3 by decreasing ΔT is possible, but the application may get freezed especially while searching for the process ID that has the priority p: 20 000 process IDs (maybe more, depending of the range allocated by Android for numbering processes) must be tested during one cycle of ΔT which requires a lot of CPU time (cf. Section V-B2).

For CC#4, we measured a throughput of 0.22 bps using $\Delta T = 800$. Decreasing ΔT is also possible for this covert channel but the configuration of the smartphone introduces a strong limitation: the waiting time for the screen to go off. Indeed, when waking up the screen for a short time to mark the beginning/ending of the transmission, the CC sender should wait that the screen goes off which usually takes several seconds. Thus, it explains that the throughput is lower than CC#3.

2) Energy consumption: Figure 6 compares the energy consumption of CC#1 and CC#2 during one minute of transmission of a byte when no user interaction occurs. It also includes the average consumption of energy of CC#3 during one minute of one hour of running. The energy consumption has been monitored using *Power Tutor* [35] during an interval of one minute of transmission.

The results shows that the priority based covert channel using the screen state detection CC#2 is more efficient than CC#1. This is probably due to the fact that the task based algorithm should scan the list of all tasks whereas the process priority algorithm only needs to check the priority of the identified process.

Regarding CC#3, the energy consumption is very high because the CC receiver waits for the CC sender between each transmission. Indeed, the CC receiver needs to scan the 20 000 PIDs in order to discover the target process that will change its priority. Worse, if the CC sender is not installed, the CC receiver will scan all the 20 000 PIDs forever, consuming a lot of energy for nothing. We measured a consumption of 60 J during one minute by the CC receiver without an active CC sender.

Finally, the graphic of Figure 6 shows that covert channels CC#1 and CC#2 have a very low footprint for a ΔT greater than 10 s compared to Android's kernel that consumes approximatively 80 mJ during a minute of transmission.

C. Defeating Covert Channel Detection Solutions

We also checked that our solution can defeat the TaintDroid tool [11] that tracks information flows[2]. For this evaluation, we read the IMEI of the phone using the CC sender and we implemented an SMS-based data leakage into the CC receiver. Therefore, we transmitted several digits of the IMEI from the sender to the receiver and sent an SMS to exfiltrate the data. TaintDroid did not report the information leak and the logs only shows the taint marking event that occurs when the CC sender reads the IMEI:

dalvikvm(673): TaintLog: addTaintFile(36): adding 0x00000400 to 0x00000000 = 0x00000400}.

Of course, when performing the two operations into the same application (reading IMEI and sending it), TaintDroid would detect the data leakage and would notify the user.

We could not evaluate our malware against *CHEX* [9] and *QuantDroid* [36] because they are not available online or not released yet. Nevertheless, we think that *CHEX* and *QuantDroid* would not notice our covert channels because *CHEX* focuses on component hijacking when exposing some components of an application and *QuantDroid* monitors IPC communication to detect implicit flows.

An approach called *XManDroid* [37] is capable to defeat covert channels in Android [30]. To achieve this goal, XManDroid inspects the inter-component communication (ICC) calls or the use of the system API in order to prevent forbidden information flows. The approach monitors the data written by an application into the system API and afterwards read by another application. We assume that XManDroid is capable of detecting at least our screen state-based covert channel and it is possible that the other covert channels will be detected as well. Nevertheless, it requires to correctly define the security policy of XManDroid for all possible covert channels. Moreover, using a too restrictive security policy may break regular applications.

VI. Discussion

In this section, we will first highlight means to counter our presented covert channels and we will afterwards discuss further means adapted from the area of network covert channels to improve the functionality of the channels.

[2]A video of this experiment is available at url2: http://dai.ly/x10lcyq. A QR code for this url can be found at the end of this paper.

A. Covert Timing Channel Limitation Means

Since we observe the process priority, the screen state, or the task list at given time intervals to realize the data transfer, our covert channels must be considered as covert timing channels. The limitation of a covert timing channel as well as its detection were discussed in the related work section but will be set in the specific context of our covert channels in this section.

1) Fuzzy Time: In 1991, Hu developed the concept of fuzzy time to limit the timing channel capacity between virtual machines in the VAX security kernel [38]. While only the kernel knows the exact current time, all virtual machines were provided with slightly different timing information. If this concept would be applied to Android, only the kernel would know the real system time but applications would be provided with slightly incorrect times in a way that it does not affect the general operation of the applications. It is also thinkable to turn of a fuzzy time for selected system applications. However, by applying the fuzzy time, the channel capacity of our screen state-based channel could be limited but the covert channel could not be completely eliminated.

Alternatively, using delays can help to reduce the covert channel throughput. In our case, requests to the process list and their priorities could be delayed. Nevertheless, if only few information (e.g., a password) shall be leaked, even a tiny channel capacity must be considered valuable for an attacker – especially since many smartphones are permanently connected to the Internet and can thus provide a slow but constant data leakage.

2) Machine Learning: While fuzzy time aims on limiting the covert channel capacity, we assume that an a posteriori detection would be feasible as well.

The detection of covert timing channels usually evaluates inter packet gaps [28], [29], i.e., the timing intervals between network packets. Since our covert channels change the usual screen behavior or process list/priority API call behavior of the Android phone, statistical differences between the normal API call behavior and the behavior with a running malware are generated. It must be assumed, that these statistical differences can be detected similarly like inter packet gap-based timing channels. However, up to now, no detection approach for the anomalies created by our channels is available.

We assume that a machine learning-based detection could be applied to the screen state behavior (e.g., using the C4.5 algorithm to create a decision tree and by providing it with recorded behavior such as screen state switches per second within a list of time slices). The C4.5 algorithm was already applied in [29] to detect covert timing channels based on the timing intervals between network packets. We will evaluate the machine learning-based detection in future work for which screen state data of smart phones from different users must be recorded in order to provide a high number of data sets for regular smart phone behavior and use. These recordings are necessary to obtain accurate results when regular behavior is compared with malicious behavior. Similarly, process list requests could be observed and C4.5 could be used to detect such process list-based covert channels.

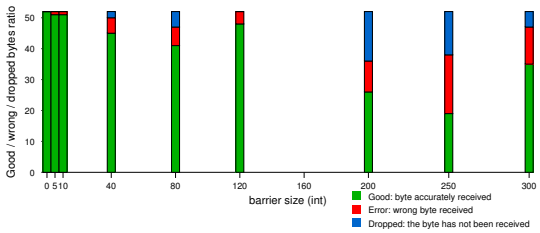

Figure 7. Transmission errors depending of the barrier size for CC#3

3) Barrier Values: In an eased version, barrier values could be used. In such a case, a covert channel alert could be generated if, for instance, more than n screen state switches per second, more than n requests to the process table, or more than n process priority requests (e.g., $n \geq 1/2$ process priority requests per second) occur over a period of i seconds (e.g., $i \geq 10$ seconds). This approach is very similar to common port scan detection algorithms which alert a system's administrator if more than n ports are scanned for a period of i seconds.

It is also thinkable to introduce fuzzy results into methods if a given barrier is reached: in order to test the effect that a barrier can have against the covert channel CC#3, we implemented a barrier into the JNI call from Process.getThreadPriority(pid). The JNI call executes the function android_os_Process_setCallingThreadPriority that we patched using the following code:

```
static int barrier = 0;
jint android_os_Process_getThreadPriority(JNIEnv* env, jobject clazz, jint pid)
{
    int barrier_size = 1; // controls barrier size
    barrier++;
    if (barrier >= 1000 && barrier < 1000 + barrier_size)
      return -1;
    if (barrier >= 1000 + barrier_size)
      barrier = 0;
...
```

This way, the variable end_of_barrier controls the amount of times a fake value is returned to the calling application. In Figure 7, we measured the effect of the barrier size on the correctness of the transmission of a message containing 52 characters (the message is "A...Za...z" and $\Delta\ T = 2s$). With $barrier\_size = 0$, the barrier is inactive. With $barrier\_size = 1$, the barrier will be activated one time over 1001 calls, which is not a big issue for CC#3 because it will probably happen during the scan of all processes. The barrier has an impact on CC#3 if the barrier is activated during the transmission. Figure 7 shows that increasing the barrier size increases the disruption of the message. Even if this countermeasure seems to easily defeat our covert channel, it is not sure that such a code could be deployed at a large scale without impacting the operations of normal applications.

B. Adaptive Covert Channel Techniques

We proposed a communication between sending and receiving applications based on time intervals and task monitoring/process priorities to provide a high-quality communication. However, it would be feasible to extend our approach to become more reliable if existing covert channel features such as internal control protocols [32], [39] or adaptive covert communication [40] would be included. The features discussed in this section are used in the context of network covert channels and this paper is the first to propose such features for covert timing channels.

Internal control protocols for covert channels are tiny protocol headers placed inside the covert information. These additional information slow down a covert channel but add sequence numbers, acknowledgement values, flags to indicate connection states, or small checksums to the covert channel. Such control protocol headers must not be represented by an area in a network packet but could also be encoded in our covert channels if the control channel is extended to encode additional values which are usually encoded in header fields (e.g., one could use multiple process IDs at the same time to represent more information).

Since covert channels do only provide limited space for meta information such as sequence numbers, Ray and Mishra implemented the *stop-and-wait automatic repeat request* (ARQ) approach for their reliable communication [32]. In ARQ, a sender waits for the acknowledgement of a packet before the next packet is sent. Ray and Mishra implemented a two bit sequence number and a two bit expected sequence number for ARQ. Using ARQ, a sequence number cannot be used twice and thus, an overrun of a sequence number is prevented although it only comprises few bits. Being easy to implement, ARQ could be realized using additional process priorities as an extended control channel for Android.

Adaptive covert channels, on the other hand, are capable of changing their communication behavior based on the current situation [40]. For instance, if a covert channel utilizes the ICMP protocol for a connection and a change in a firewall configuration results in blocked ICMP packets, an adaptive covert channel is capable to switch to another network protocol (e.g. IRC, HTTP, or a streaming protocol) to bypass the firewall. In the context of Android, multiple covert channel techniques could be utilized simultaneously in order to bypass future protection means. For instance, if the data channel is blocked, the application will – if full-duplex – receive no response and thus, can try using another data channel to bypass the new protection means.

VII. Conclusion

We presented novel covert channels for Android. Our covert channels overcome existing protection means like TaintDroid. Although the bandwidth of our presented channels is low in comparison to existing covert channels, the quality of the data transmission is high since we split our covert channels into a control channel to start and stop a transaction, and a data channel that transfers the actual hidden information. To minimize the raised attention, our channels comprise a low energy consumption, minimize disruptions of the smartphone use, and require only few privileges in the Android system.

The experimental results indicate that a good malware design helps to obtain a low energy consumption during the transmission which is important in order to achieve stealthiness. Choosing the right way of implementing the covert channel is also important to utilize as less Android permissions as possible. Two of our covert channels do not require any Android permission in order to work.

We discussed further means to counter our timing channels as well as we proposed means from the area of network covert channels to enhance the presented local covert channel. Future work will add adaptive covert channel techniques to our approach as well as it will evaluate the use of a machine learning-based covert channel detection.

Acknowledgment

This work was supported by the French National Agency (ANR) through the "Ingénierie Numérique et Sécurité" Program (Project LYRICS n°ANR-11-INSE-0013).

References

[1] I. Asrar, "Android Threat Tackles Piracy Using Austere Justice Measures," *Irfan Asrar's blog*, 2011.

[2] B. Morrow, "BYOD security challenges: control and protect your most sensitive data," *Network Security*, vol. 2012, no. 12, pp. 5–8, Dec. 2012.

[3] D. Barrera and P. Van Oorschot, "Secure Software Installation on Smartphones," *IEEE Security & Privacy Magazine*, vol. 9, no. 3, pp. 42–48, 2011.

[4] C. Orthacker, P. Teufl, S. Kraxberger, A. Marsalek, J. Leibetseder, and O. Prevenhueber, "Android Security Permissions – Can we trust them?" in *3rd Int. ICST Conf. on Security and Privacy in Mobile Information and Communication Systems (MOBISEC 2011)*, vol. 3, 2011, p. 12.

[5] S. Mansfield-Devine, "Android architecture: attacking the weak points," *Network Security*, vol. 2012, no. 10, pp. 5–12, Oct. 2012.

[6] L. Batyuk, M. Herpich, S. A. Camtepe, K. Raddatz, A.-D. Schmidt, and S. Albayrak, "Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within Android applications," in *6th International Conference on Malicious and Unwanted Software*. IEEE Computer Society, Oct. 2011, pp. 66–72.

[7] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak, "An Android Application Sandbox system for suspicious software detection," in *5th International Conference on Malicious and Unwanted Software*. IEEE, Oct. 2010, pp. 55–62.

[8] M. Graa, N. Cuppens-boulahia, and A. Cavalli, "Detecting Control Flow in Smarphones: Combining Static and Dynamic Analyses," in *The 4th International Symposium on Cyberspace Safety and Security*. Melbourne, Australia: Springer, Dec. 2012, pp. 33–47.

[9] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: statically vetting Android apps for component hijacking vulnerabilities," in *ACM conference on Computer and communications security - CCS '12*. Raleigh, NC, USA: ACM Press, 2012, p. 229.

[10] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "DroidMat: Android Malware Detection through Manifest and API Calls Tracing," *2012 Seventh Asia Joint Conference on Information Security*, pp. 62–69, Aug. 2012.

[11] 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 *9th USENIX Symposium on Operating Systems Design and Implementation*. Vancouver, BC, Canada: USENIX Association, Oct. 2010, pp. 393–407.

[12] M. Nauman, S. Khan, and X. Zhang, "Apex: extending Android permission model and enforcement with user-defined runtime constraints," in *5th ACM Symposium on Information, Computer and Communications Security*. Beijing, China: ACM Press, Apr. 2010, pp. 328–332.

[13] G. Russello, B. Crispo, E. Fernandes, and Y. Zhauniarovich, "YAASE: Yet Another Android Security Extension," in *Third IEEE International Conference on Information Privacy, Security, Risk and Trust*. MIT, USA: IEEE Computer Society, Oct. 2011, pp. 1033–1040.

[14] C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun, "Analysis of the communication between colluding applications on modern smartphones," in *28th Annual Computer Security Applications Conference*. Orlando, Florida, USA: ACM Press, Dec. 2012, pp. 51–60.

[15] A. Armando, A. Merlo, M. Migliardi, and L. Verderame, "Would you mind forking this process? a denial of service attack on android (and some countermeasures)," in *SEC*, ser. IFIP Advances in Information and Communication Technology, D. Gritzalis, S. Furnell, and M. Theoharidou, Eds., vol. 376. Springer, 2012, pp. 13–24.

[16] C. Mulliner, "Privacy leaks in mobile phone internet access," in *Proc. 14th International Conference on Intelligence in Next Generation Networks*, Berlin, Germany, October 2010, pp. 1–6.

[17] P. Stirparo and I. Kounelis, "The MobiLeak project: Forensics methodology for mobile application privacy assessment," in *Proc. 7th International Conference for Internet Technology and Secured Transactions*, London, UK, 2012, pp. 297–303.

[18] B. W. Lampson, "A note on the confinement problem," *Commun. ACM*, vol. 16, no. 10, pp. 613–615, 1973.

[19] S. Zander, G. J. Armitage, and P. Branch, "A survey of covert channels and countermeasures in computer network protocols," *IEEE Communications Surveys and Tutorials*, vol. 9, pp. 44–57, 2007.

[20] Z. Li, A. Goyal, and Y. Chen, "Honeynet-based botnet scan traffic analysis," in *Botnet Detection*, 2008, pp. 25–44.

[21] J. C. Wray, "An analysis of covert timing channels," in *Proc. 1991 Symposium on Security and Privacy*. Oakland, CA: IEEE Computer Society, 1991, pp. 2–7.

[22] C. H. Rowland, "Covert channels in the TCP/IP protocol suite," *First Monday*, vol. 2, no. 5, May 1997.

[23] N. B. Lucena, G. Lewandowski, and S. J. Chapin, "Covert channels in IPv6," in *5th Int. Workshop on Privacy Enhancing Technologies*, ser. LNCS. Cavtat, Croatia: Springer, 2005, vol. 3856, pp. 147–166.

[24] R. Accorsi and C. Wonnemann, "Detective information flow analysis for business processes," in *Business Processes, Services Computing and Intelligent Service Management*, ser. Lecture Notes in Informatics, vol. 147, Leipzig, Germany, 2009, pp. 223–224.

[25] R. A. Kemmerer, "Shared resource matrix methodology: an approach to identifying storage and timing channels," *ACM Transactions on Computer Systems*, vol. 1, no. 3, pp. 256–277, 1983.

[26] P. A. Porras and R. A. Kemmerer, "Covert flow trees: A technique for identifying and analyzing covert storage channels," in *1991 IEEE Symp. on Security and Privacy*, Oakland, CA, 1991, pp. 36–51.

[27] M. H. Kang and I. S. Moskowitz, "A pump for rapid, reliable, secure communication," in *Proceedings of the 1st ACM Conference on Computer and Communication Security*, 1993, pp. 119–129.

[28] V. H. Berk, A. Giani, and G. V. Cybenko, "Detection of covert channel encoding in network packet delays," Department of Computer Science - Dartmouth College, Tech. Rep., 2005.

[29] S. Zander, "Performance of selected noisy covert channels and their countermeasures in ip networks," Ph.D. dissertation, Centre for Advanced Internet Architectures, Swinburne University of Technology, 2010.

[30] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang, "Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones," in *Network and Distributed System Security Symposium*. San Diego, California, USA: The Internet Society, Feb. 2011.

[31] H. Kim, J. Smith, and K. G. Shin, "Detecting energy-greedy anomalies and mobile malware variants," in *The 6th international conference on Mobile systems, applications, and services*. Breckenridge, Colorado, USA: ACM Press, Jun. 2008, pp. 239–252.

[32] B. Ray and S. Mishra, "A protocol for building secure and reliable covert channel," in *Proceedings of the Sixth Annual Conference on Privacy, Security and Trust (PST 2008)*. Fredericton, New Brunswick, Canada: IEEE, 2008, pp. 246–253.

[33] S. Wendzel and J. Keller, "Systematic engineering of control protocols for covert channels," in *Proc. 13th Conf. on Communications and Multimedia Security*, ser. LNCS, vol. 7394. Springer, 2012, pp. 131–144.

[34] R. L. Solso and J. F. King, "Frequency and versatility of letters in the english language," *Behavior Research Methods & Instrumentation*, vol. 8, pp. 283–286, 1976.

[35] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, ser. CODES/ISSS '10. ACM, 2010, pp. 105–114.

[36] T. Markmann, D. Gessner, and D. Westhoff, "QuantDroid: Quantitative approach towards mitigating privilege escalation on android," in *IEEE International Conference on Communications*, Budapest, Hungary, 2013.

[37] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi, "XManDroid: a new android evolution to mitigate privilege escalation attacks," TU Darmstadt, Tech. Rep. TR-2011-04, 2011.

[38] W.-M. Hu, "Reducing timing channels with fuzzy time," in *1991 Symposium on Security and Privacy, IEEE Computer Society*, Oakland, CA, 1991, pp. 8–20.

[39] D. Stødle, "Ping tunnel – for those times when everything else is blocked," 2009.

[40] F. V. Yarochkin, S.-Y. Dai, C.-H. Lin, Y. Huang, and S.-Y. Kuo, "Introducing P2P architecture in adaptive covert communication system," in *First Asian Himalayas International Conference on Internet, 2009. AH-ICI 2009*, Kathmandu, Nepal, 2009, pp. 1–7.

url1: 

url2: