Report

Application Collusion Attack on the Permission-Based Security Model and its Implications for Modern Smartphone Systems

Author(s):
Marforio, Claudio; Francillon, Aurélien; Capkun, Srdjan

Publication Date:
2011

Permanent Link:

https://doi.org/10.3929/ethz-a-006720730 →

Application Collusion Attack on the Permission-Based Security Model and its Implications for Modern Smartphone Systems

Claudio Marforio, Aurélien Francillon, Srdjan Capkun
Department of Computer Science, ETH Zurich, Switzerland
{*maclaudi, afrancil, capkuns*}*@inf.ethz.ch*

***Abstract*—We show that the way in which permission-based mechanisms are used on today's mobile platforms enables attacks by colluding applications that communicate over overt and covert communication channels. These attacks allow applications to indirectly execute operations that those applications, based on their declared permissions, should not be able to execute. Example operations include disclosure of users private data (e.g., phone book and calendar entries) to remote parties by applications that do not have direct access to such data or cannot directly establish remote connections. We further show that on today's mobile platforms users are not made aware of possible implications of application collusion–quite the contrary–users are implicitly lead to believe that by approving the installation of each application independently, based on its declared permissions, they can limit the damage that an application can cause. In this work, we show that this is not correct and that application permissions should be displayed to the users differently (e.g., in their aggregated form), reflecting their actual implications. We demonstrate the practicality of application collusion attacks by implementing several applications and example covert channels on an Android platform and an example channel on a Windows Phone 7 platform. We study free applications from the Android market and show that the potential for application collusion is significant. Finally, we discuss countermeasures that can be used to mitigate these attacks.**

I. Introduction

Today's smartphone operating systems allow users to install third-party applications directly from on-line *application markets*. In order to perform their functions, applications typically need specific permissions such as network access or access to user's personal data. However, given the large number of independent developers, every application cannot be trusted to behave according to their declared purpose. Certain types of malicious behaviors can be detected by inspection and testing while others cannot, malicious applications therefore find their way into the application markets [13], [34], [35], [36].

To limit the potential impact of malicious applications, mobile phone operating systems (e.g., Android OS [4], Symbian OS [11], MeeGo OS [9], Windows Phone 7 [7]) implement a permission-based security model[1] that restricts the operations that each application can perform. In this model, applications are explicitly given permissions for operations that they need. Example permissions include network access, access to user's private data, etc. The permission model is enforced by the operating system, which runs each application process in a sandbox, thus limiting the operations of each process.

Upon installing the applications, the user is made aware of the permissions that the applications request. A user can then refuse to install an application that requests permissions which do not correspond to its declared functions. E.g., a weather forecast application will likely need network access, but should not require access to user's contacts; a contacts organizer application will need access to the user's contacts, but not to the network. This model allows users to limit the impact that malicious applications, if installed, have on their systems and on their personal privacy. In the above examples, even if the applications act maliciously, but independently, they will not be able to leak user's personal data since these applications can either not access personal data or cannot communicate it to third parties. Given this, it is typically assumed that if users are careful enough in installing applications, thus granting the requested permissions–and if the operating system properly enforces those permissions–that users will be able to effectively protect their privacy [33].

In this work we show that this assumption is not correct and that the permission-based security model cannot fully protect user's privacy under colluding applications. We show that colluding applications can be constructed on today's mobile platforms and can use covert as well as overt channels to aggregate their permissions. Colluding applications can therefore indirectly execute operations that those applications individually, based on their permissions, should not be able to execute. E.g., if the colluding weather forecast and contact

[1]It is also sometimes described as a *capability model*.

organizer applications communicate, they will be able to leak user's personal data to third parties since their aggregated permissions allow it.

We further show that on today's mobile platforms that implement the permission-based model users are not made aware of possible implications of application collusion–quite the contrary–users are implicitly lead to believe that by approving the installation of each application independently, based on its declared permissions, 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 maybe most severe for mobile platforms–these platforms are designed for personal use and are designed to facilitate the installation of third-party applications.

Even existing security products, such as [16], that are used for the analysis of application permissions, analyze and report the permissions independently for each individual application and therefore do not take into consideration application collusion. Given application collusion, those products do not correctly reflect the privacy implications of the applications that the users install.

We note that application collusion attacks on the permission-based model are not a result of a software vulnerability and are not related to a particular implementation–they are a consequence of the basic assumption on which the permission-based model relies: that 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 thus can be exploited to break the permission-based model.

We demonstrate that attacks using malicious application collusion can be easily implemented on today's mobile platforms. We mainly focus on the Android OS, where we implement several colluding applications that communicate over a number of overt and covert channels, including those realized using broadcast intents, process execution times, process enumeration and thread enumeration. We further show that the user does not need to install two colluding malicious applications on his device for this attack to be successful. A single malicious application with access to user's data will still be able to leak this data by passing it (over a covert channel) to a script executed within the phone's browser. We study free applications from the Android market and show that the potential for application collusion is significant: a large number of application pairs and individual applications can violate user's privacy using covert communication channels. Finally, we demonstrate application collusion on a Windows Phone 7 platform where we implement an overt channel using the Windows Media Library.

To mitigate these attacks, we discuss ways how overt and covert channels can be either partially or fully closed. While overt channels can be discovered and restricted by using taint analysis (e.g. TaintDroid [22]), by reducing access to some APIs or by better sandboxing, other–mainly covert channels–cannot be blocked without serious degradation of system performance. This is not surprising since covert channels have long been known to be hard to entirely prevent on real systems [20], [32]. Given this, permissions should be granted and managed under the assumption that applications can aggregate their permissions by collusion over covert channels. This is, however, not the case on current mobile phone operating systems, where before installing an application the users are shown only the permissions of to-be-installed application–but they are not explicitly made aware of the aggregated permissions of possibly colluding applications. This makes it difficult for the users to understand the implications of installing an application and typically leads to an underestimation of the associated risks. To address this problem, besides proposing ways of closing covert channels, we further propose several simple measures that can be implemented by the operating systems to help users and organizations limit and better manage the risks associated with installing untrusted applications.

In summary, the contributions of this paper are the following: (1) We show that the restrictions, which are placed on the operations of the applications in the permission-based security model, can be overcome by colluding applications. This allows colluding applications to indirectly, over overt and covert channels, execute operations that those applications–based on their permissions–should not be allowed to execute. (2) We demonstrate the practicality of this attack by implementing several example channels on Android and Windows Phone 7 platforms; we further discuss how some of these channels can be either partially or fully closed. (3) We describe several realistic scenarios in which users can become victims of this attack. (4) We study free applications from the Android market and–although in our study we do not identify any colluding applications–we show that the potential of application collusion is significant. (5) We draw conclusions on the security and use of the permission-based security model.

II. Permission-Based Access Control Model under Application Collusion

In this section, we first describe the permission-based model used by smartphone operating systems and we then introduce application collusion attacks on the systems that use this model.

A. Permission-Based Access Control

A smartphone is a personal device with a single user that owns the phone and all the private data that is stored on the phone. The goal of access control mechanisms on the phones is therefore not to protect a user from other users, but to protect the user (and their data) from the applications and to protect applications from other applications. The assets that need protection are user's private data stored on the phone (calendars, contacts, access credentials, etc.), the data that the phone records (GPS location, microphone, etc.) and resources that it provides (ability to call, send text messages, etc.). Modern mobile platforms rely on a modular application model where, apart from the base of the system, each functionality can be provided by an independent application. Those applications can be simple and require no special permissions (e.g. a simple game) or can be complex and require an extended set of access permissions.

Given that applications for mobile platforms are developed by independent and untrusted third parties, they need to be contained in order to prevent malicious behavior. In addition to the typical operating system inter-process and user protections, one of the main goals of the security mechanisms in such an environment is therefore to control access to resources by applications and prevent user's private data to be abused (e.g., modified, ransomed [10] or disclosed to third parties without authorization).

In this environment the major focus is on multilateral security, in particular, unless explicitly permitted, applications should not be able to share private data. UNIX, on which many mobile operating systems [4], [1] are based, traditionally relies on discretionary access control (DAC). The model which these OSs apply for application permissions is a Mandatory Access Control (MAC) [19]. While MAC is well suited to prevent unauthorized data flows, DAC is less constraining and allows uncontrolled communication between applications. For example, an application can write data to a file and make it available to another application ID.

Modern smartphone OSs can be divided in three categories with respect to access control to sensitive data or resources:

- with security-enforcement mechanisms for third party applications [4], [9]
- with centralized application reviews before publication on the market [1], [6]
- with both security-enforcement mechanisms and centralized reviews for third-party applications [11], [8], [7]

Most of these Operating Systems ([6], [4], [11], [8]) allow *by default* installation of third party applications from non centralized sources (i.e. sources not controlled by the company developing the OS, such as the web). This practice is known as *sideloading*. Only iOS [1] and Windows Phone 7 [7] enable installation of applications *only* through a centralized, tightly controlled, store.

We will focus on Android [4] and Windows Phone 7 [7] smartphone operating systems which implement the following mechanisms to contain untrusted applications:

- Permission-based access control: to limit which resources one application can access.
- Application containment (or sandboxing): to prevent overt information flow from one application to another.

We note that the iOS case–where there is no permission-based security model implemented–is not relevant for our scenario, an attacker has therefore no advantage in using application collusion.

B. Permissions Enforcement

Permission enforcement is a crucial part of the permission-based model. The first line of defense is application sandboxing. Application sandboxing relies on process containment where an application is unable to tamper with the environment of another application. Looking, for example, at the Android operating system, application containment is performed at several levels:

- UNIX file access rights, enforced by the Linux kernel. This allows applications to protect their data in a private directory. This is achieved through reuse of UNIX user and group IDs. In particular each application, at installation time, is given its own UNIX user ID which is the ID under which it will run and under which file access will be granted or denied.
- UNIX user groups, enforced by the kernel. To enable access to data or resources, the application can then be assigned to one or more system group IDs. Those group IDs will often be used to control access to device descriptors or socket files, e.g., access to the network (used similarly to POSIX capabilities).

- Middleware (e.g., Java API level on Android) where a security kernel provides the permission verification mechanism. This level also allows applications to define custom permissions.

Additionally, each application on the Android operating system runs under its own Java Virtual Machine: *Dalvik*. However, the choice of the Java environment was apparently not dictated to achieve isolation since any Android application can embed its own native code in the form of a Java Native Interface (JNI) library. This choice was probably done for portability, developer acceptance of the language and a reduced usage of memory unsafe languages. The availability of native code does not by itself allow one application to bypass permissions since native code is still executed in the "user ID" sandbox. It also does not provide access to other applications files and resources for which it has no permissions (e.g., in the form of a group ID)[2].

C. Application Collusion on Smartphones

While the model presented in the previous section is very efficient in preventing individual applications from bypassing their permissions, it is fragile against multiple applications. We show this by introducing the concept of *colluding applications*.

Colluding applications are those applications that cooperate in a violation of some security property of the system. These applications do not need to individually break any security permission or abuse software vulnerabilities. They instead use existing or construct new *communication channels* to perform actions or access resources to which they would independently be unable, e.g., due to their respective permission restrictions.

The attack by colluding applications is possible because current security mechanisms are not focused on controlling the channels that two applications can use to communicate. Instead, most efforts have been made to achieve application containment or sandboxing. This is likely due to the fact that tight information flow control (with overt or covert channels) is typically of little concern on personal computers OSs, on which many smartphone operating systems are based.

We show that application collusion can lead to disclosure of user's personal data. We thus show that the use of application sandboxing is ineffective as a way of solving the confinement problem.

Covert Channels on Smartphones: In order to collude, applications need to communicate. We describe a covert channel as any channel or method that is intentionally used by two applications to communicate while it was not intended to be used for communication. In [29] covert and overt channels are described as follows.

> The channels, such as buffers, files, and I/0 devices, are *overt* because the entity used to hold the information is a data object; that is, it is an object that is normally viewed as a data container.
>
> *Covert channels*, in contrast, use entities not normally viewed as data objects to transfer information from one subject to another. These nondata objects, such as file locks, device busy flags, and the passing of time, are needed to register the state of the system.

Finally, [29] notes that overt channels need to be *controlled by enforcing the access control policy* while covert channels needs to be dealt with using dedicated methods.

Covert and overt channels can be found at all the levels of abstraction in a system, we define three levels at which these channels can be found in current smartphones:

- **High.** The level of the API that an operating system provides to the developers. In the Android operating system this is given by the Java API and in Windows Phone 7 operating system by the C#/Silverlight APIs. This is the *simplest* level, where covert channels may be more easily closed.
- **OS.** This is the level of the operating system which is exposed through native calls that will exploit information present in the operating system. We believe that this level may be closed in some cases, but could potentially severely damage backward compatibility (or in some cases even be impossible to close).
- **Low.** The hardware level which is exposed through exploiting hardware functionalities of the smartphone. It is thus highly dependent on the hardware specifications of that particular smartphone model. Those channels may not be closed without severe performance degradation of the system.

Different levels usually also imply different bandwidth. In particular we notice that bandwidth is usually directly proportional to the level, with higher bandwidth associated to *high*-level covert channels.

D. Implications of Application Collusion

We outline a scenario that illustrates possible uses of application collusion for information leakage. We consider the following fictitious applications that we use throughout the paper: *Weather*, with access to the

[2]It however makes it easier to use system or kernel level exploits [34], but those are not considered in this work.

network, displays the weather information, and *ContactsManager*, which organizes user's contacts and has only access to the contacts. Another colluding application could be a *PasswordsManager* which would not disclose private data stored on the phone but passwords that the user would store believing that they are safe.

Each application is limited to accessing data or resources for which it has received permissions. The user, knowing that such security enforcement is in place installs both applications without hesitating. Without the user noticing, the two applications are *colluding*, using a communication channel to exchange information and therefore eluding the operating system security. We describe below two possible consequences: privacy disclosure and social engineering.

Privacy Disclosure: ContactsManager could send contacts information to a given remote server by sending the contact information, over a covert channel, to the *Weather* application. Similarly the *PasswordsManager* could send the stored passwords to the *Weather* application. *Weather* would simply take the data, for which it did not have any permission, and send it by using its legitimate Internet permission. Figure 1 depicts the described example.

Social Engineering: Weather could fetch some information from the Internet, using its permission to access that resource, and pass it to the *ContactsManager*. Subsequently the *ContactsManager* could store the received information as a contact on the user's phone book or replace an existing number, all in accordance with the permissions granted to it by the system. The user, upon receiving a call or any other communication from the maliciously added contact would more likely accept it since it would appear as coming from a "known source".

We also note that, while we analyze the problem of two applications exchanging data, collusion could exist between multiple applications or could be used to enable specific actions on the phone (e.g., microphone activation, making calls, sending SMS, phone reboot).

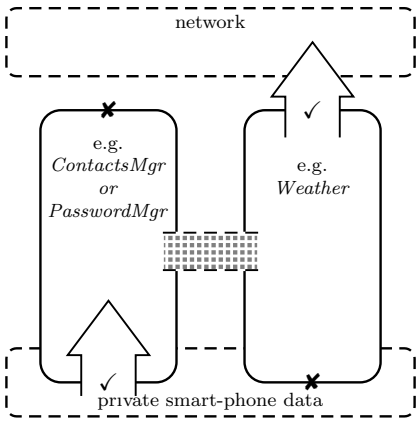


Figure 1. The *ContactsManager* or *PasswordsManager* application on the left and the *Weather* application on the right colluding through a covert communication channel. The ContactsManager does not have access to the network, but has access to user's contacts. The Weather application has no access to user's contacts but can access the network. The ContactsManager leaks user's contacts to the Weather application, which then sends this information to a third party.

III. Application Collusion in Android OS

In the Android OS, the permission-based security model is used as follows:

1) The developer explicitly states which permissions the application requires for functioning.
2) At install time the user is shown a list of permissions required by the application and a brief general description.
3) If the user accepts to install the application given the displayed permissions, the application is downloaded and installed.
4) At run-time the operating system enforces that the application doesn't access any data or resource for which it does not have a permission.

Permissions are given in the form of strings saved in a *manifest* XML file, supplied by the developer and present in every Android application. Once granted, the permissions are immutable; if a combination of permissions is found to be unsuitable to the user after the installation, the application needs to be manually removed.

The way that the permission-based security model is used in Android assumes users can correctly judge the implications of permissions that applications request. Although this might not be true for some users, increasing numbers of users do understand the risks of different types of access (e.g., to their personal data and to the network [16]). Figure 2 shows the Android permissions of three applications, presented to the user at their installation time. Here, the permissions of applications (a) and (b) should not seem threatening to the user since each application only request either access to personal data or to the network. Permissions of application (c), however, might raise suspicion of the user since they include both access to the network and to the user's private data. The user might therefore be inclined not to install application (c) whereas he would feel that it is safe to install applications (a) and (b). The argument in favor of this security model is that the user, knowingly granting permissions to one application to access sensible data and the Internet, would only do

so upon checking beforehand the genuineness of the application and that of the operating system to enforce that appropriate restrictions apply. We note that it is widely believed that if an application does not require the permission to access the Internet it would not pose any privacy disclosing threat [36].

As we already discussed in Section II, overt and covert channels can allow applications to collude and communicate across their sandboxes. The applications are thus able to aggregate their permissions. In Android OS, application collusion is not considered and the users will still believe that it is safe to e.g., install applications (a) and (b) from the above example, whereas these applications might collude and disclose the users private data.

We proceed by exploring available overt and covert channels that can be used by the application and we construct example colluding applications for the Android OS.

A. Colluding Applications

We focus on colluding applications that can be used to leak user's private data to third parties. We use our example applications from Section II (ContactsManager and Weather), and we build these applications for the Android OS. Here, the *ContactsManager* application only has the `READ_CONTACTS` permission, and the *Weather* application only has the `INTERNET` permission. We assume that the user has installed both applications and thus granted them these permissions. We further assume that these applications were built by the same developer or by colluding developers and collude to leak user's contact information to a third party. When colluding applications are used for data leakage we classify them either as "sources", denoting a class of applications that has a permission to access private data, or as "sinks", representing a class of applications that can receive and forward the data to third parties. Figure 1 depicts two colluding "source" and "sink" applications.

B. Overt and Covert Channels in Android

Table I lists a set of overt and covert channels that we found in the Android operating system at different levels as described in Section II. Seven of those channels have been successfully implemented and tested. Furthermore, Table I shows which channels can be closed without impacting existing applications (according to the results of the analysis we report in Section IV). We also mark channels that could be detected by information flow tracking such as TaintDroid [22]. TaintDroid relies on dynamic taint tracing to analyze information flow at run-time. By tainting private data, TaintDroid discovers applications that disclose private data. We mark the channels that could be detected with this system in Table I, however, other channels that do not explicitly transfer data are not likely to be prevented by employing such a system.

In the covert channels literature, low-bandwidth covert channels are generally considered low risk, because they can carry only small quantities of information. However, user's location privacy can be fully violated by transmitting a 64 bit encoding of their GPS coordinates.

Implementation of the described covert channels (excluding the *Cache Collision*) is fairly straightforward. We provide snippets of code for three covert channels in order to show the simplicity of implementing such covert channels by colluding applications.

Broadcast Intent: Figure 3 shows the Java code needed by the *source* and *sink* applications to send and receive a broadcast that contains private data.

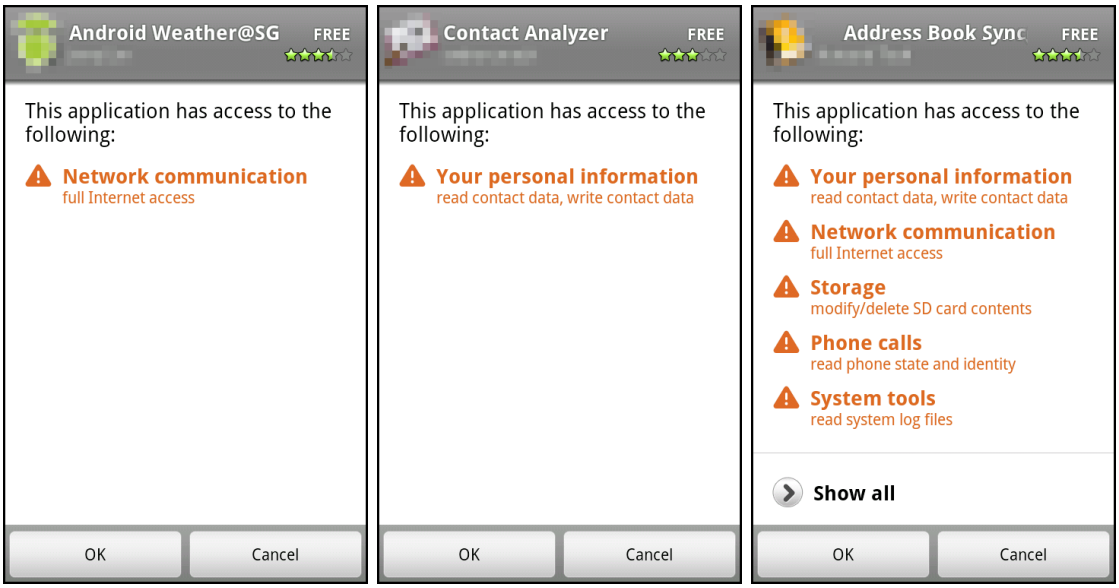
Processes Enumeration from API: Figure 4 shows the Java code needed to exchange information through the number of running services. By using this method the *source* will, for a given amount of time t, either spawn an extra service or not. The *sink* will keep on counting the number of services that the *source* application is using and will infer information from that. To use the Java API to list the running service the *sink* application will require the extra `GET_TASKS` permission.

Processes Enumeration from native code: Figure 5 shows how the *sink* could find the number of forked processes of *source* through the native (C) code. In this covert channel the *source* application forks its process from a native JNI function using a `fork()` call, while the *sink* application counts how many processes have been forked by parsing the `/proc/` file system. A similar approach can be used by using `pthread_create()` from the native code instead of forking the process on the *source* side. The *sink* in this case could simply read the file `/proc/<PID>/status` to count the number of spawned threads. This allows information to be exchanged between the *source* and *sink* applications.

The main intent of showing these snippets of code is to show that it is easy to actually use the listed (Figure I) communication channels in practice.

C. Colluding Application and the Browser

We now extend the proposed concept of colluding applications and consider the scenario in which there is only one application that wants to send private data to a third party web service without requesting the



(a) Application with only Network access. (b) Application with only Contact access. (c) Possibly suspicious Application.

Figure 2. The screens presented to the users at application installation time. Applications declare which permissions they require and the user then decides whether to install the application. The figure shows the permissions of three example applications taken from the Android Market (note that we do not want to imply that these applications are malicious)

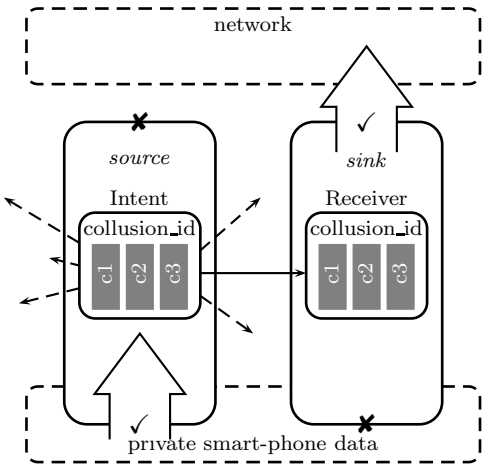
Channel	Implemented	Can be Closed *	Detectable by Tainting	Requires Extra Permissions	Bandwidth
Overt Channels					
Shared Permissions	✓	✓	✓	no	high
Broadcast Intents	✓	✓	✓	no	high
SDCard Shared Files	✓	✘	✓	WRITE_EXTERNAL_STORAGE	high
Covert Channels					
Processes Enumeration from API	✓	✘	✘	GET_TASKS	low
Processes Enumeration from Native Code	✓	✘	✘	no	low
Threads Enumeration from Native Code	✓	✘	✘	no	low
Timing Attack	✘	✘	✘	no	low
Cache Collision [39], [37], [15]	✘	✘	✘	no	low
One application and Browser					
Timing Attack	✓	✘	✘	no	low

Table I

SUMMARY OF OVERT AND COVERT CHANNELS THAT WE DISCOVERED IN THE ANDROID OS. THE CHANNEL LISTED IN THE LAST ROW OF THE TABLE IS A SPECIAL CHANNEL BECAUSE FOR IT TO BE USED, IT IS NOT REQUIRED THAT TWO INSTALLED APPLICATIONS COLLUDE, BUT INSTEAD IT CAN BE USED TO LEAK PRIVATE INFORMATION FROM AN INSTALLED APPLICATION TO A SCRIPT RUNNING IN A BROWSER. (* *can be closed* COLUMN LISTS THE CHANNELS THAT CAN BE CLOSED WITHOUT IMPACTING EXISTING APPLICATIONS.)

permissions to connect to the network. In this case the application could use a timing covert channel by either executing some CPU-bound code or not (i.e. to send a 1 or a 0 to the web service). It is sufficient then to have a colluding web page which also tries to execute CPU-bound instructions (i.e. through JavaScript) and times each execution. The JavaScript application will notice a substantial difference in execution time whether the colluding application installed on the device is executing its operations or not.

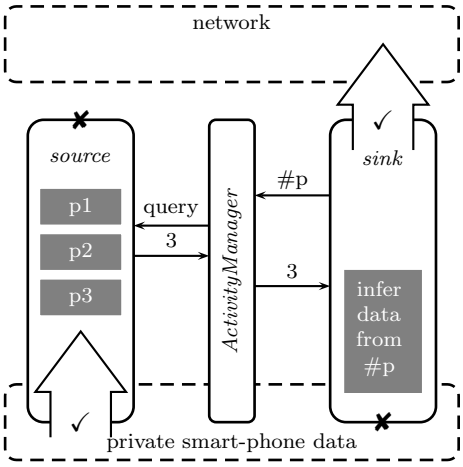
We have implemented and tested this proposed covert channel by making the application and the JavaScript code execute dummy RC4 encryption and decryption operations. Furthermore, the user does not have to navigate to a colluding web page. The installed application can open a web browser page at any time, for example, in the night when the phone has long been in an idle state. The possibility to use the Browser as a way to send private data has been described [31] but this proposed method could be easily detected by TaintDroid. Our




```
Context ctxt = this.getApplicationContext();
Intent i = new Intent("colluding-id");
i.putExtra("contacts", contacts);
ctxt.sendBroadcast(i);
```

```
public class AlarmReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context ctxt, Intent intent) {
        Log.i(TAG, "contacts: " + intent.getExtras()
            .getString("contacts"));
    }
}
...
IntentFilter intentF = new
    IntentFilter("colluding-id");
this.getApplicationContext()
    .registerReceiver(alarmReceiver, intentF);
```

Figure 3. The figure on the left shows the visualization of the *Broadcast Intent* overt channel. The right figure shows the Java source code that implements the *Broadcast Intent* channel used by the *source* (on the top) and by the *sink* (on the bottom).




```
Intent svc = new Intent(this, AppOneService.class);
startService(svc);
...
stopService(svc);
```

```
ActivityManager am = (ActivityManager) this
    .getApplicationContext()
    .getSystemService(Context.ACTIVITY_SERVICE);
List<ActivityManager.RunningServiceInfo> services = am
    .getRunningServices(20);
for (ActivityManager.RunningServiceInfo rsi : services) {
  if (rsi.process.equalsIgnoreCase("package.id")) {
    Log.i(TAG, "Found: " + rsi.process);
  }
}
```

Figure 4. The figure on the left shows the visualization of the *Process Enumeration* covert channel. The right figure shows the Java source code that implements the *Process Enumeration* channel used by the *source* (on the top) and by the *sink* (on the bottom).

proposed channel–while being low-bandwidth–is also much harder to detect.

D. Summary

We have described the idea of colluding applications. Furthermore, we implemented, tested and described different overt and covert channels that colluding applications can use. We also described a scenario where only one application can use the browser to send data to a third-party service.

Finally, while a recent proposal [22] suggests to use tainting to track the flow of data and enforce some privacy preserving rules, this solution would still be unsuitable to protect against some combinations of application collusion and covert channels we presented.

IV. Analysis of the Applications From the Android Market

To assess the potential for application collusion, we analyzed the permissions of the applications in the Android Market [3]. We aimed at identifying those applications that could violate the user's privacy if their permissions are aggregated. We then analyzed the Android applications themselves. For this purpose, we collected the function calls made to shared libraries which could be used for the construction of covert channels.

Our test set consisted of 9127 randomly selected applications, which represent roughly 10% of the total number of available applications in the Android Market [2]. Code analysis was then focused on a subset of

```
char * proc = "/proc/";
char * cmdline = "cmdline";
char * target = "package.id.of.source.app";
dp = opendir (proc);
if (dp != NULL) {
  while (ep = readdir (dp)) {
    file_name = malloc(snprintf(NULL, 0, "%s%s/%s", proc, ep->d_name, cmdline) + 1);
    sprintf(file_name, "%s%s/%s", proc, ep->d_name, cmdline);
    fp = fopen (file_name, "r");
    if (fp != NULL) {
      char contents[250];
      fgets(contents, 249, fp);
      if (strcmp(contents, target) == 0) // we found a target
        counter++;
      fclose(fp);
    }
    free(file_name);
  }
  (void) closedir (dp);
}
return counter;
```

Figure 5. The figure shows the C source code used by the *sink* to detect the number of running processes spawned by the *source*. The code lists the directories in `/proc` and looks for those that contain the *source* package id as their `cmdline` content.

the 6513 freely downloadable applications.

A. Potential for Application Collusion

We start by giving a few examples on how developers can lead users to download two colluding applications. While this goes partly beyond the scope of this paper, and is related to social engineering, we believe that some examples will stress the need to protect against application collusion attacks.

The simplest way to have two colluding applications installed is, having one application already present on the device, to show (possibly fake) advertisements pointing to the second application in an appealing way. Another possibility arises, for example, when a company buys a second one which develops other applications. The buying company could then submit an update to one application as the user will not be prompted again with the permissions screen but will instead have the new application installed automatically. Another possibility could be that the colluding application resides in the SDK provided by third parties, such as advertisement companies. In this case any developer, by embedding the provided code, could embed code that could potentially misbehave.

In order to analyze the potential for privacy violation, we categorized the applications of the test set, according to their declared permissions, into the two following groups (described in Section III):

- **Sources**: applications that have permissions to read private data but cannot share data with third parties. In the Android OS we identify sources by having at least one of the following permissions but none of the permissions of the *sinks*: `ACCESS_COARSE_LOCATION`, `ACCESS_FINE_LOCATION`, `PROCESS_OUTGOING_CALLS`, `READ_CALENDAR`, `READ_CONTACTS`, `READ_FRAME_BUFFER`, `READ_HISTORY_BOOKMARKS`, `READ_INPUT_STATE`, `READ_LOGS`, `READ_OWNER_DATA`, `READ_PHONE_STATE`, `READ_SMS`, `RECEIVE_SMS`, `RECEIVE_MMS`, `RECORD_AUDIO`.
- **Sinks**: applications that do not have permissions to read private data but have some way to communicate to third parties or otherwise change private data on the phone. In the Android OS we identify sinks by having at least one of the following permissions but none of the permissions of the *sources*: `INTERNET`, `WRITE_CONTACTS`, `WRITE_HISTORY_BOOKMARKS`, `GET_TASKS` [3].

Our analysis shows that there are 223 possible *sources* and 2786 possible *sinks*. The number of sources times the number of sinks gives us the number of possible pairs of applications that could collude in order to exchange information: $223 * 2786 = 621278$.

[3] `GET_TASKS` is a special case, this permission can be used to mount a covert-channel between two applications. In the analysis we considered as *sinks* application with the `GET_TASKS` and at least another sink-only permission.

B. Inspecting Applications for Covert Channels

Having identified possible *sources* and *sinks* we then performed a basic static analysis over downloaded packages.

Native Library Analysis: The Android Native Development Kit (NDK) allows to ship custom shared libraries (`.so` files) within an application package (`.apk`), these native libraries are then loaded by the Dalvik virtual machine as custom JNI libraries.

We extracted a total of 237 independent shared libraries used by 202 applications out of the downloaded 6513. We then disassembled the libraries to analyze which were calling the `fork()` or `pthread_create()` functions. We found that 11 libraries, used by 12 applications, were using `fork()`. 28 libraries, used by 24 applications, were using `pthread_create()`. This shows that a small percentage of applications are using shared libraries (3%) and that between such applications 11.8% are using POSIX threads and 5.9% are creating child processes.

We then analyzed the ASCII strings in extracted libraries to find those that access the `/proc` folder. To understand why the proc file system was accessed, we manually inspected the disassembled library code. Visual inspection showed that the applications were not performing any kind of collusion but apparently were extracting information on processes and CPU usage for legitimate uses.

API Call Analysis: While information flow techniques (such as TaintDroid [22] would in many cases detect API-level covert channels, we still performed some basic analysis of the downloaded *source* applications.

In order to analyze the API calls of applications, we decompiled them from Java `.class` to Java sources as follows:

1) extracted the `classes.dex` file, that contains the compiled class files, from the `.apk` application package.
2) converted the `classes.dex` file to plain java bytecode using the `dex2jar` [12] tool.
3) decompiled the resulting `.class` files using the JD-GUI [21] tool.

From the resulting Java source code it was possible to search for patterns of calls that could be used for the construction of covert channels listed in Table I.

For this purpose we performed a set of basic string-matching searches[4] with the following patterns:

- `'Uri.parse("http'` this would show all packages that would open an "http" or "https" page. We followed the links and visually inspected the landing pages for malicious JavaScript or other scripting code.
- `'getSharedPreferences.*,1'`, `'getSharedPreferences.*,2'` and `'getSharedPreferences.*,[a − z]'` to discover packages that use *shared Preferences* with a `MODE_WORLD_READABLE` or `MODE_WORLD_WRITABLE` attributes. This would lead to unchecked information passing between two applications. We then visually analyzed the code following these calls to check which variables were set in the preferences file.
- `'getExternalStorageDirectory()'` this would show all packages accessing the SD Card for reading or writing to files. We run this test to have an estimate of how many packages were using the SD Card. Similarly, we did that at the permissions level and showed that 1764 packages were requesting such permission.
- `'createPackageContext('` this would show if a package is trying to create a `Context` object from another package name. None of the sources was exhibiting such a malicious intent.
- `'putExtra('` this would show source application that tries to pass information to another through Broadcast Intents with extra information attached to them. We then manually checked the extra contents passed and the Intent receiver. This analysis also showed no sign of application collusion.

Summary: The result of the analysis makes us believe that, considering the set of analyzed application packages, application collusion techniques are not currently employed by applications present on the Android Market. However, a potential of such collusion is there: 2.5% applications could act as *sources* and 43.9% could act as *sinks*.

V. Application Collusion on Windows Phone 7

In this section we describe an implementation of the application collusion attack in the Windows Phone 7 operating system.

A. The Windows Phone 7 Platform Security

Windows Phone 7 is a recent smartphone operating system from Microsoft; it's third party application support has been completely redesigned from previous version.

It features a relatively tight sandboxing: applications are only installable from a centralized "market", the

[4]Using the Linux `grep` and `egrep` tools.

MarketPlace, after a review and certification process carried out by Microsoft. A set of permissions are enforced at execution time according to the manifest file present in every application, `WMAppManifest.xml`. Permissions are displayed during the application installation. However, the user's consent is explicitly requested only for a few permissions, such as for using the location of the device.

Third party developers are limited to using managed *.NET* code: native code is not allowed and the API is limited to a reduced set of features. Applications are only active when they are in the foreground and it is impossible to run code in background[5]. Alarm and event call-backs are unavailable when the application is not in foreground. Moreover, applications can not access the file system structure other than for an application-specific *isolated storage*. At the moment of writing it is not known if Microsoft will add any kind of multitasking to its Windows Phone 7 OS through a software update and it is not known whether this decision is made to preserve battery life or for addressing security issues.

Applications cannot access personal data directly, instead such data is indirectly made available through choosers and launchers. For example, when an application needs to access an email address, the application will use the *EmailAddressChooserTask*. This will call the built-in *Contacts* application that will prompt the user to select a contact in the contacts list, the result will be returned to the calling application.

While this architecture allows a tight control over access to data, it is also very limiting what applications can achieve. For example, in such an environment it is impossible to provide a third party backup service or calendar notifications. It is therefore likely that the API will evolve over time.

A consequence of limited private data access and an environment without multitasking is that it reduces the impact of malicious applications and the attack surface for application collusion. However, in what follows we describe a realistic application collusion attack we implemented and tested on a Windows Phone 7 device.

B. Colluding Applications on Windows Phone 7

Since multitasking is not available in the current state of the API, application communication channels need to be found in persistent inter-applications areas, such as the storage system. In particular, the operating system does not allow applications to share files and does not expose the file system.

[5]However, the system supports multitasking for applications from Microsoft and device manufacturers.

We now describe a covert channel that can be used to pass information between applications: the image gallery can be accessed by colluding applications and does not require any user interaction for read and write operations. The only requirements for the applications is to specify the permission to access the Media Library: `<Capability Name="ID_CAP_MEDIALIB"/>`.

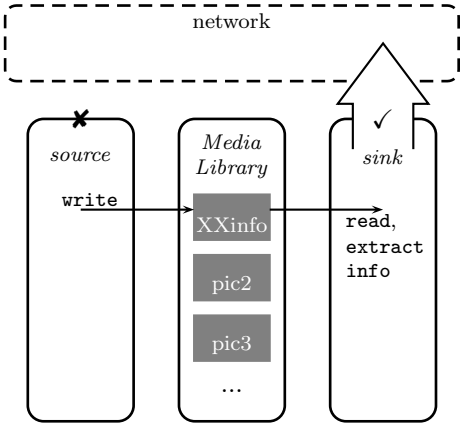
Given that access to private data is limited with the current state of the API we can imagine the *source* application to be a Password Manager that the user could use to store passwords for different services and protect them with a "master" password. The *source*, as for the other examples, would not have access to the Internet and so the user feels safe that his passwords would not be shared to third parties.

The *source* application saves an image in the media library with the information that it wants to transmit to the *sink* application encoded in the title of the image. The *sink* application browses the media library searching for particular patterns in the images titles and extracts the intended information. This communication channel is effective because the user is never exposed to the titles of the images, even when images are shared through an email message, the original image name is substituted by a dummy name. Moreover, by saving black images the colluding applications can further conceal their communications to the user as those would not be visible in the gallery. Figure 6 shows the described communication channel and displays the relevant code.

The described covert channel is high bandwidth and is hardly detected by the user. Nonetheless taint analysis would easily detect this channel. A lower bandwidth channel could potentially be used: the *source* application saves a number n of images in the library to send $log_2(n)$ information to the *sink*. The *sink* then needs to count the number of images and produce the encoded message. This is a noisy (other applications might save images) and low bandwidth (the user needs to manually start both applications) channel but it would nonetheless allow covert communications. We note that this covert channel would not be identifiable by tainting but it could potentially be easier to spot by the user. Finally, if future versions of the API allow applications to also delete files from the Media Library then both presented channels would be harder to detect by the user, and thus could become more efficient.

VI. Mitigation Techniques

Mitigating application collusion can be made either by reducing access to sensitive APIs or by limiting communication possibilities.




```
myFileStream = myStore.OpenFile(tempJPEG,
    FileMode.Open, FileAccess.Read);
MediaLibrary library = new MediaLibrary();
Picture pic = library.SavePicture(
    "XXPrivateInfoNotToBeDisclosed.jpg",
    myFileStream);
myFileStream.Close();
```

```
MediaLibrary library = new MediaLibrary();
IEnumerator<Picture> pics = library.Pictures
    .GetEnumerator();

while (pics.MoveNext()) {
    if (pics.Current.Name.Substring(0,2) == "XX") {
        System.Diagnostics.Debug
            .WriteLine(pics.Current.Name);
    }
}
```

Figure 6. Application collusion in Windows Phone 7. The figure on the left shows the visualization of the overt channel using the Media Library. In this case there is no access to shared data from the phone, but we can imagine the *source* to be a Password Manager and therefore containing itself private data that should not be disclosed. The figure on the right shows the C# source code used by the *source* (on the top) to write an image into the Media Library with a title used for collusion and by the *sink* (on the bottom) for detecting images that pass information through the Media Library.

Reducing applications access to private data by for example involving user action on each data access or transfer helps mitigating the impact of colluding applications (and more generally malicious applications). However, this also limits what applications can perform; in such an environment it is impossible for an application to perform automated backups of private data.

There is thus a tension between the available features provided to application developers and the containment of malicious applications.

More generally, solving the confinement problem, and in particular closing all possible covert channels in a system, is known to be a hard problem [20], [32]. It is even more so in the case of smartphones, where both performance and the possibility of installing arbitrary applications from a large pool of developers are keys to winning the users and developers adoption of a given platform.

In what follows we describe general purpose techniques that help in mitigating application collusion and solutions to some specific attacks presented in the paper.
General Purpose Techniques: We provide a list of techniques that we believe should be adopted by all smartphone operating systems:

- Application Collusion Aware Design: the threat of colluding applications should be considered during design of new systems. In particular:
  Limit Attack Surface: when tailoring APIs exposed to third-party developers, carefully ponder decisions that can expose trivially-implementable communication channels between applications
  Limit Multitasking: covert channels that result from competition for access to resources (CPU time, cache and bus contention) can be strongly limited by disallowing multitasking. Again this limits the diversity of applications that can be implemented on the system.
- Application Review: before publication of applications on the markets, the application should be reviewed with specific techniques to detect application collusion such as discussed in Section IV.
- Aggregate Permissions: present to the users an aggregate of permissions from all (or from a subset of less-trusted applications) installed on the device whenever a new application is installed. This could help the user to know the worst-case scenario if some applications would be colluding. Another possibility would be to inform the user of other applications currently running on the system and their permissions upon starting a new application. These approaches might be confusing for the user and need to be designed in an usable fashion (e.g., by not aggregating the permissions of the applications that are trusted or company-generated).
- Static Analysis: perform some method of basic or more involved static analysis at review time. Knowing potential communication channels utilized by applications by using a tool or a set of techniques to statically analyze the application being reviewed searching for known channels implemented by the applications.
- Policy-Based Installation Strategy: this 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.

API Level Channels: We review application collusion attacks on the Android OS that we described in Section III-B and propose solutions to these attacks while keeping in mind that harshly limiting application's abilities is not advisable. We start by considering the API-level channels.

- The creation of a *Shared Preferences* channel requires the two colluding applications to set their preferences as `MODE_WORLD_READABLE`. Since none of the applications we analyzed in Section IV used this feature, we believe that this mode can be removed from the API without imposing limitations to most developers.
- The creation of a *Broadcast Intent* channel is possible because the API does not force the developers to specify a receiver or to specify the permissions the receiver needs to hold in order to be able to receive the broadcast message. A solution that would close this channel would be to enforce that the receiver has at least the same permissions as the one of the sender[6].
- The *SDCard Shared Files* channel is similar to the *Shared Preferences* channel. To close this channel, it would suffice to prevent two applications from sharing files. The applications would then have to explicitly request permissions to share files arbitrarily, which should trigger the user's attention at install time.
- The *Process Enumeration from API* channel is somehow similar to the *SDCard* channel and can be similarly prevented because it requires an extra permission to be requested at install time.

The first three channels described above could be detected by TaintDroid [22] and can be discovered either through static analysis or at run time (unless obfuscation is used). The last, *Process Enumeration from API* channel, is a more subtle channel than other API-level covert channels and it could be equally implemented using a lower-level function which would allow a direct interaction with the operating system and so would not require any extra permission. This attack would be difficult to detect using known techniques.

Operating System Level Channels: Covert channels that can be established at this level are not detectable by information flow analysis (e.g. TaintDroid [22]) and their prevention requires further investigation.

The OS-level channels that we presented use native calls (`fork()` and `pthread_create()`) to exchange data and are thus more difficult to detect than API level channels. Our analysis shows that few applications actually use a library with the mentioned native calls. To close the *Process Enumeration from Native Code* and *Threads Enumeration from Native Code* channels, one possibility would be to remove such functionality from the ones provided to developers (or detect their use before accepting applications on the *Market*). This solution is not preferable because it might limit what one application could or could not achieve. For example, applications may create threads or spawn new processes to achieve better responsiveness of the user interfaces or to achieve better performances for complex computations (e.g. for games). The *GR-Security* [5] Linux extension restricts access to the */proc/<PID>/* directory and this can help in reducing the channel bandwidth. However, the number of running processes is still available to applications, which would still allow the creation of a covert channel.

Hardware Level Channels: The *Cache Collision* covert-channel and the *CPU Usage* (e.g. timing) attacks are the hardest to remove without serious performance degradation or functionality impact. Solution to those (such as preventing multitasking or flushing caches between process scheduling) would limit the overall performance, reactivity or power consumption of the system. Furthermore, data tainting and common information flow control is ineffective in this scenario. Closing these channels requires novel approaches, which we leave as future work.

Windows Phone 7: As mentioned already, the Windows Phone 7 platform offers very limited attack surface. Nonetheless, we described an implemented covert channel through the *Media Library*. We note that some presented countermeasures are implemented in the Windows Phone 7: Multitasking or Native Code Execution are absent from the APIs available to developers.

Finally, since applications run *.NET* managed code that is executed in a virtual machine, we believe that a tainting mechanism as the one described in TaintDroid [22] could also be implemented on the Windows Phone 7 platform with reasonable performance impact.

VII. Related work

In this section, we review related work.

The Confinement Problem and Covert Channels: The confinement problem was first described by Lampson in [30] as the problem of preventing unauthorized

[6]This problem can also lead to elevation of privileges during exploitation of software vulnerabilities [19].

communication, over overt or covert channels, between two subjects on a system. It was generally recognized to be a difficult problem in practice, for example in [20] Denning and Denning states :

> One type of flow cannot be controlled easily, if at all. A program can convey information to an observer by encoding it into some physical phenomenon without storing it into the memory of the computer. These are called flows on covert channels [LAMP73, LIPId75]. A simple covert channel is the running time of a program. [...] Cost-effective methods of closing all covert channels completely probably do not exist.

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 [38], [25], 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 [26].

Software covert channels can be mitigated by a careful analysis of the usage of visible and alterable variables used by system calls [42] or using a formal model for analyzing programs [41] using a semi-automated technique. However, hardware related covert channels (e.g., timing, competition to access a bus or a cache, paging) are difficult to prevent and recent processor designs have been shown to increase covert channels [44].

As an example, multi-core application processors are already available for smartphone devices, which would render covert channels over cache highly reliable [44]. Possible mitigation techniques include using fuzzy time [28] and preventing multitasking.

Permission-Based Security: The security architecture of Android is presented in [27] while that of Windows Phone 7 is presented in [7]. A significant amount of work has been performed on the android platform and specifically on the permission-based model [43], [34], [19], [36], [18], [17], [14]. In [14] 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. The *Kirin* tool [23] uses predefined security rules 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 [36] allows run-time control over communication between applications according to their permissions. In [17], Burns discusses possible unchecked information flows due to applications that use *Broadcast Intents* without proper permissions checking.

Some of the channels we describe here are explicit (overt channels) and could therefore be detected by information flow tracking tools such as TaintDroid [22] or SCanDroid [24]. Because they track explicit data flows in the Java bytecode, they cannot discover or prevent information passed through covert channels, information flow obfuscation or information flows in native code.

Soundcomber[40] is a proof of concept malware for android smartphones, it uses the microphone to harvest sensitive information, such as credit card numbers, by detecting voice and tone patterns. Independently of our work, Soundcomber uses covert and over channels used to exfiltrate sensitive data. The channels presented are using globaly available settings (vibration, volume, screen lock, etc.) or file locks, those are however limited to colluding applications that can execute in parallel, e.g. on platforms that allow multitasking. In contrast to the use of covert channels in Soundcomber, we describe the problem of colluding applications comprehensively on multiple platforms.

Finally, [45] aims to use existing methodologies to detect covert channels in the Linux kernel, however it is not clear if those results are complete or usable in practice.

VIII. Conclusion

We showed that permission-based mechanisms used on today's operating systems for mobile devices such as Android OS and Windows Phone 7 are vulnerable to attacks by colluding applications. We demonstrated how these attacks allow applications to indirectly execute operations which those applications, based on their permissions, should not be able to execute. Specifically, we showed that application collusion can lead to disclosure of user private data to remote parties. We further studied free applications from the Android market and showed that the potential of application collusion is significant. Finally, we discussed countermeasures that can be used to mitigate application collusion attacks.

References

[1] Apple : iOS (up to version 4.1). http://developer.apple.com/iphone.

[2] Google : 100000 Android Applications in Market. http://twitter.com/#!/AndroidDev/status/28701488389.

[3] Google : Android Market. http://www.android.com/market/.

[4] Google : Android OS (up to version 2.2). http://developer.android.com/.

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

[6] HP : WebOS. http://developer.palm.com.

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

[8] Nokia : Maemo OS (up to version 5). http://maemo.org/development/.

[9] Nokia : MeeGo OS. http://meego.com/developers/meego-architecture/meego-architecture-layer-view.

[10] Ransomware (malware). From Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Ransomware_%28malware%29.

[11] Symbian OS. http://www.symbian.org/.

[12] dex2jar. http://code.google.com/p/dex2jar/, 2010.

[13] Jonathan Anderson, Joseph Bonneau, and Frank Stajano. Inglorious Installers: Security in the Application Marketplace. In *Workshop on the Economics of Information Security (WEIS)*, 2010.

[14] David Barrera, Üne, Paul C. van Oorschot, and Anil Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *CCS '10: Proceedings of the 17th ACM conference on Computer and Communications Security*, pages 73–84, New York, NY, USA, October 2010. ACM.

[15] Daniel J. Bernstein. Cache-timing attacks on AES, 2005.

[16] 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/.

[17] Jesse Burns. Developing Secure Mobile Applications for Android. https://www.isecpartners.com/files/iSEC_Securing_Android_Apps.pdf, 2008.

[18] Avik 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.

[19] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege Escalation Attacks on Android. In *ISC 2010: Proceedings of the 13th Information Security Conference*, October 2010.

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

[21] Emmanuel Dupuy. Jd-gui 0.3.3. http://java.decompiler.free.fr/?q=jdgui, 2010.

[22] William Enck, Peter Gilbert, Byung gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[23] William Enck, Machigar Ongtang, and Patrick 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.

[24] Adam Fuchs, Avik Chaudhuri, and Jeffrey Foster. Scandroid: Automated security certification of android applications, 2011. In preparation, available at http://www.cs.umd.edu/~avik/projects/scandroidascaa/.

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

[26] Virgil 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.

[27] Google. *Android : Security and Permissions*. http://developer.android.com/guide/topics/security/security.html.

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

[29] Richard A. Kemmerer. A practical approach to identifying storage and timing channels: Twenty years later. In *Proceedings of the 18th Annual Computer Security Applications Conference*, ACSAC '02, pages 109–, Washington, DC, USA, 2002. IEEE Computer Society.

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

[31] Anthony McKay Lineberry. These aren't the permissions you're looking for. BlackHat USA, August 2010.

[32] Steven B. Lipner. A comment on the confinement problem. In *Proceedings of the fifth ACM symposium on Operating systems principles*, SOSP '75, pages 192–196, New York, NY, USA, 1975. ACM.

[33] Patrick McDaniel and William Enck. Not so great expectations: Why application markets haven't failed security. *IEEE Security and Privacy*, 8:76–78, 2010.

[34] Jon Oberheide. Android Hax. SummerCon 2010, June 2010. Available at http://jon.oberheide.org/files/summercon10-androidhax-jonoberheide.pdf.

[35] Jon Oberheide and Farnam Jahanian. When mobile is harder than fixed (and vice versa): demystifying security challenges in mobile environments. In *Proceedings of the Eleventh Workshop on Mobile Computing Systems and Applications*, HotMobile '10, pages 43–48, New York, NY, USA, 2010. ACM.

[36] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically rich application-centric security in android. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 340–349, Washington, DC, USA, 2009. IEEE Computer Society.

[37] Colin Percival. Cache missing for fun and profit, May 2005.

[38] Fabien A.P. Petitcolas, Ross J. Anderson, and Markus G. Kuhn. Information hiding-a survey. *Proceedings of the IEEE*, 87(7):1062 –1078, July 1999.

[39] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *CCS '09: Proceedings of the 16th ACM conference on Computer and Communications Security*, pages 199–212, New York, NY, USA, 2009. ACM.

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

[41] Alan B. Shaffer, Mikhail Auguston, Cynthia E. Irvine, and Timothy E. Levin. A security domain model to assess software for exploitable covert channels. In *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, PLAS '08, pages 45–56, New York, NY, USA, 2008. ACM.

[42] Chii-Ren Tsai, Virgil 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.

[43] Troy Vennon and David Stroop. Threat Analysis of the Android Market. Technical report, June 2010. Smobile systems technical report, Available at http://threatcenter.smobilesystems.com/.

[44] Zhenghong Wang and Ruby B. Lee. Covert and side channels due to processor architecture. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 473–482, Washington, DC, USA, 2006. IEEE Computer Society.

[45] Gaoshou Zhai, Yufeng Zhang, Chengyu Liu, Na Yang, MinLi Tian, and Hengsheng Yang. Automatic identification of covert channels inside linux kernel based on source codes. In *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human*, ICIS '09, pages 440–445, New York, NY, USA, 2009. ACM.