Kharon dataset: Android malware under a microscope

Nicolas Kiss, Jean-François Lalande, Mourad Leslous, Valérie Viet Triem Tong

Kharon dataset: Android malware under a microscope

N. Kiss
EPI CIDRE
CentraleSupelec, Inria, Univ. Rennes 1, CNRS,
F-35065 Rennes, France

J.-F. Lalande
INSA Centre Val de Loire
Univ. Orléans, LIFO EA 4022,
F-18020 Bourges, France

M. Leslous, V. Viet Triem Tong
EPI CIDRE
CentraleSupelec, Inria, Univ. Rennes 1, CNRS,
F-35065 Rennes, France

Abstract

Background – This study is related to the understanding of Android malware that now populate smartphone's markets. **Aim** – Our main objective is to help other malware researchers to better understand how malware works. Additionally, we aim at supporting the reproducibility of experiments analyzing malware samples: such a collection should improve the comparison of new detection or analysis methods. **Methodology** – In order to achieve these goals, we describe here an Android malware collection called Kharon. This collection gives as much as possible a representation of the diversity of malware types. With such a dataset, we manually dissected each malware by reversing their code. We run them in a controlled and monitored real smartphone in order to extract their precise behavior. We also summarized their behavior using a graph representations of the information flows induced by an execution. With such a process, we obtained a precise knowledge of their malicious code and actions. **Results and conclusions** – Researchers can figure out the engineering efforts of malware developers and understand their programming patterns. Another important result of this study is that most of malware now include triggering techniques that delay and hide their malicious activities. We also think that this collection can initiate a reference test set for future research works.

1 Introduction

Android malware have become a very active research subject in the last years. Inevitably, all new propositions of detection, analysis, classification or remediation of malware must deal with their own evaluation. This evaluation will rely on a set of "malicious indicators" that have to be detected/analyzed/classified as bad and a set of "legitimate indicators" that have to be ignored by the evaluation method. Designing a set of "good things" appears simple but on the contrary, for precise evaluation, the set of "bad things" should be perfectly understood. We claim here that rigorous experiments have to rely on malware samples totally reversed.

Building an understandable dataset to be used for dynamic analysis is a difficult challenge. Indeed, an automatic methodology for reverse engineering a malware does not exist. First, no mature reverse engineering tool has been developed for Android that would be comparable to the ones used for x86 malware. Second, each malware is different and finding automatically the malicious code by statically analyzing the bytecode is a very difficult task because this code is mixed up with benign code. It requires a human expertise to extract relevant parts of the code. Finally, most advanced malware now include countermeasures to avoid to trigger their malicious behavior at first run and in emulated environments. Thus, an additional expertise is required to understand the special events and conditions the malware is awaiting.

Thus, building an understandable malware dataset requires a huge amount of work. We made this effort for evaluating our previous works [1] and we propose here to make our training dataset well documented in order to initiate the construction of a reference dataset of Android malware. Our goal is to build a well documented set of malware that researchers can use to conduct reproducible experiments. This dataset tries to represent most of the possible know types of malware that can be found. When choosing a malware for representing a type, we excluded the malware that are too obfuscated or encrypted to be reversed engineered in a reasonable time.

The contributions of the paper are:

1. A precise description of the internals of 7 malware samples i.e. how each malware attacks the operating system, how it interacts with external servers and the effects from the user perspective;

2. A graphical view of the induced information flows when the malware is successfully executed;

3. Instructions on how to trigger the malware in order to make reproducible the attacks operated by each malware of the dataset. These instructions are essential for conducting experimental evaluation of methodologies that analyze dynamic events.

In the following, we give an overview of existing Android malware datasets and present online services dedicated to malware analysis. In Section 3, we put seven malware under a microscope and give a precise description of each of them. Section 4 concludes this article.

2 Related works

2.1 Android security basics

Android security relies on standard Unix and Java security paradigms that are the base of standard Linux distributions. Android processes are isolated from each others using different Unix process ids but applications can still communicate between each other, by using so called *Intents* that transport exchanged information. Application are executed by a virtual machine or compiled by a Ahead-of-time compiler. Both mechanisms include runtime checks for implementing security and the most important security checks are guaranteed by the Linux kernel itself. For example, the access to the network is provided using a dedicated *inet* group.

A special file, called the Manifest, declares the software components that are the possible entry points for the application. The most important components can be: an *Activity* i.e. a set of graphical components for composing a screen of an application; a *Service* that can be run in or outside the main process of the application and has no graphical representation; a *BroadcastReceiver* that executes the declared callback when the application receives information. For starting or making these components communicate, *Intent*s are Java Objects that provide facilities to transport data. Some pre-defined *Intent*s encode some system events. For instance, the *Intent BOOT_COMPLETED* notifies applications that the smartphone has finished the boot process. As malware need to communicate, we often observe the use of *Intent*s.

The security policy of an application is expressed by the permissions declared in the Manifest. Permissions can protect resources (network, data, sensors, etc.) or system data or components (list of processes, ability to keep the smartphone awake, etc.). Other advanced security mechanisms can be found in recent Android versions such as checking the boot sequence integrity or the use of SELinux for enforcing mandatory policies at kernel level.

2.2 Malware datasets

One of the most known dataset, the Genome Project, has been used by Zhou et al. in 2012 to present an overview of Android malware [19]. The dataset is made of 1260 malware samples belonging to 49 malware families. The analysis was focused on four features of Android malware: how they infect users' device, their malicious intent, the techniques they use to avoid detection and how the attacks are triggered. The last feature is the most interesting for us as we want to provide a dataset that can be easily used by people working on dynamic analysis of Android malware. According to Zhou et al. analysis, Android malware can register for system events to launch their attack e.g. the `BOOT_COMPLETED` event sent when the smartphone is up. In addition to system events, some malware directly hijacks the main activity or the handler of the user interface components.

Unfortunately, the exact condition required to execute the malicious code is never provided by the paper's authors. For example, for the case of DroidKungFu1, we know that the malicious code can be launched at boot time but there is no indication about the time bomb used to schedule the execution of the malware. Indeed, DroidKungFu1 executes its malicious code only when 240 minutes have passed and this condition is checked by reading a specific value in the application preferences. Without this information, a dynamic analysis fails to observe interesting behaviors.

In [4], Arzt et al. present FlowDroid, a static taint analysis tool for Android applications. The goal of FlowDroid is to detect data leaks in Android applications using static analysis. To evaluate their tool, Arzt et al. have developed DroidBench[1], a set of applications implementing different types of data leakage thanks to implicit information flows, use of callbacks and reflections. Applications in this dataset are classified according to the technique they use to leak data and contain a description of the leak performed by the application. For each application, the source code for performing the leak is given, which makes result comparison and evaluation easy. At the time of writing, the DroidBench repository contains 120 applications of which APK and source code are both available. As it only performs a data leak, the source code is really minimalist. Unfortunately, the applications in DroidBench are not real malware samples and are only meant to evaluate dynamic and static analysis tools. They do not provide a dataset with the complexity of real malware in which benign code is mixed with malicious code.

Contagio dataset is a public collection of Android malware samples [15]. It was created in 2011 and is regularly updated, which makes it one of the most up to date public dataset. Each contribution is published as an article on the blog associated to the dataset with a link to

download the samples, a description, and an external link generally to a VirusTotal report. The static analysis part of the report seems to be done with Androguard and provides different information such as the required permissions, the components, the use of reflection, cryptography, etc. The dynamic analysis part lists the observed behavior during the execution: started services, accessed files, use of sensitive functions and connection to remote servers. Such information gives an insight on the nature of the application but is useless to determine how to launch the malicious code of a malware sample.

2.3 Online services

Some analysis services are provided online for commercial or research purposes. They are mainly developed to detect Android malware and potentially harmful applications but can also give a better understanding of an application.

One of them, Verify Apps, is the service used by Google to scan applications submitted on Google Play and applications installed on users' devices. Google does not provide any technical detail on their service but according to their report on Android security for 2014 [8], their tool uses a mix of static analysis and dynamic analysis. The goal of the analysis is to extract multiple features of the application and decide if it is potentially harmful by comparing these features with the ones used by known malware. For instance, the service compares the developer's signature with known signatures that are associated with malicious developers or malicious applications. The report provided by Google gives an insight on the type of security threats but lacks details on how these threats are executed. Unfortunately, the results of the analysis are not publicly available which makes the service not useful for research purposes.

Andrubis [12] is an online service that analyzes Android applications statically and dynamically to detect malicious behaviors using a combination of TaintDroid [7], Androguard, apktool and have analyzed more than 1,000,000 applications. Lastly, VirusTotal is an online scanning platform that uses 54 antiviruses and 61 online scan engines to perform analysis on files uploaded on its web page or sent by email. It uses several tools to perform the analysis, such as Androguard to disassemble and decompile APK packages, and Cuckoo sandbox to dynamically analyze an execution.

We claim that these platforms give very basic information and are not sufficient to understand deeply malware. We believe that every research team conduct apart their own reverse analysis. It is a huge amount of work which is often redone and thus has to be gathered and published. Thus, the description of a malware dataset is a complementary approach to online analysis tools.

Table 1: Malware of the Kharon dataset

Malware	Description SHA 256 hash value	Known Samples
BadNews [16]	Remote administration tool (Contagio) 2ee72413370c543347a0847d71882373c1a78-a1561ac4faa39a73e4215bb2c3b	15
SimpleLocker [13]	Ransomware (Contagio) 8a918c3aa53ccd89aaa102a235def5dcffa04-7e75097c1ded2dd2363bae7cf97	1
DroidKungFu [10]	Remote admin. tool (Genome project) 54f3c7f4a79184886e8a85a743f31743a0218-ae9cc2be2a5e72c6ede33a4e66e	34
MobiDash [6]	Agressive adware (Koodous) b41d8296242c6395eee9e5aa7b2c626a2-08a7acce979bc37f6cb7ec5e777665a	4
SaveMe [11]	Spyware (Contagio) 919a015245f045a8da7652cefac26e71808b2-2635c6f3217fd1f0debd61d4330	1
WipeLocker [5]	Data eraser (Contagio) f75678b7e7fa2ed0f0d2999800f2a6a66c717-ef76b33a7432f1ca3435b4831e0	1
Cajino [18]	Spyware (Contagio) 31801dfbd7db343b1f7de70737dbab2c5c664-63ceb84ed7eeab8872e9629199	4

3 Seven malware under a microscope

In this section, we present a detailed analysis of seven malware. We randomlly studied a lot of malware (more than 30) and selected the ones that were not too obfuscated or using ciphering techniques. We choosed recent ones that have been known to have been widespread on user's smartphones. These seven malware cover most of the known types of malware [19]: Aggressive adware, Fee paying services malicious usage, Ransomware, Remote Administration Tool, Spyware and Data Eraser. When studying each malware candidate for representing a type, we excluded the malware that are too obfuscated or encrypted to be reversed engineered in a reasonable time. We followed the advices of Rossow et al. [17] in order to support any future experiments: the dataset is balanced, cleaned and studied in a controlled sandbox (*correctness*); the experiment setup and malware list is documented (*transparency*); malware are executed in a real smartphone and sufficiently stimulated (*realism*); chosen malware have no network spread capabilities (*safety*).

For each malware presented in Table 1, we indicate its provenance in order to help researchers to rebuild the dataset[2]. We conducted a two step analysis in order to precisely describe their malicious code and their triggering condition. In a first part, we have manually reversed the bytecode and inspected it. This static analysis helps us to locate where the malicious code is and learn how it can be triggered. In a second part we did a dynamic analysis, triggered the previously identified malicious code and thus monitored all the malicious be-

haviors. We performed the experiment on a Nexus S with Android 4.0 *Ice Cream Sandwich* to which we added AndroBlare (further details below). We rooted our device and installed the Superuser[3] application if the application requires root privileges. Our monitoring process consists in dynamically tracking where information belonging to the analyzed sample spread during its execution and then building what we call a System Flow Graph to observe the malicious behavior of Android malware [2]. The information flow tracking is done thanks to AndroBlare[4], a tool that tracks at system level the information flow between system objects such as files, processes and sockets. The produced directed graph represents the observed information flows: it is a compact and human-readable representation of the observed malware activities captured by AndroBlare. The vertices are the information containers such as files and the edges are the information flows observed between these information containers.

3.1 BadNews, a remote administration tool

Badnews [16] is a remote administration tool discovered in April 2013. Its malicious final charge depends on commands received from a remote server. The malicious code is located in the package *com.mobidisplay.advertsv1*. Its behavior can be divided into three distinguished stages:

Stage 1: Malicious service setup and sensitive data recovery. Badnews starts at the reception of the `BOOT_COMPLETED` intent or the `PHONE_STATE` intent. When one of these intents is received, the service *AdvService* is started. On creation, this service collects information about the device such as the IMEI, the device model, the phone number and the network operator. Finally, this service sets up an alarm manager that is in charge of broadcasting an intent for the receiver *AReceiver*. This receiver will restart *AdvService* every four hours with an intent containing an extra data named *update* and set to true.

```
final AlarmManager aM = this.getApplicationContext().
    getSystemService("alarm");
final PendingIntent broadcast = PendingIntent.getBroadcast((
    Context)this, 0, new Intent(this, AReceiver.class),
    134217728);
aM.cancel(broadcast);
final long elapsedRealtime = SystemClock.elapsedRealtime();
aM.setRepeating(3, elapsedRealtime, 14400000L, broadcast);
```

Stage 2: Notify the C&C server of the availability of the device. Badnews transforms the device into a slave of a C&C server located at *http://xxxplay.net/api/adv.php*[5]. When *AdvService* is restarted with the extra data *update* set to true, it creates a thread which executes a function named *getUpdate()*. This function contacts the server and begins with sending an HTTP post request with the sensitive information collected on creation.

Stage 3: Execute the service order. The function *getUpdate()* then receives an answer from the server, which contains one of the following orders: 1) Open an URL; 2) Create a notification with an URL to open; 3) Install a shortcut that will open an URL; 4) Download and install an APK file 5) Create a notification with an APK file to download and install 6) Install a shortcut that will download an APK; 7) Update the primary or secondary server address. The APK files that might be installed are potentially malicious. During our observations, the server sent a malicious version of Doodle Jump and a fake version of Adobe Flash that seems to be a game.

Triggering Condition. As the malware requires a server to obey commands, we built a fake server and forged the commands. For example, for implementing a fake install command of another malware (*malware2.apk*), we create a file *index.html* containing:

```
{"status" : "install",
 "sound" : 0,
 "vibro" : 0,
 "apkname" : "Malware2",
 "url" : "http://192.168.0.10/malware2.apk"}
```

where *192.168.0.10* is the address of the local computer. We serve this file using a python web server. For substituting the server url in badnews.apk, we unpack the APK, substitute *http://xxxplay.net/api/adv.php* by *192.168.0.10/index.html*, repack it again, and sign the new APK:

```
$ apktool d badnews.apk # Then edit the smali files
$ apktool b badnews -o new_badnews.apk
$ jarsigner -verbose -keystore ~/.android/debug.keystore -
      storepass android -keypass android new_badnews.apk
      androiddebugkey
```

Then we install the new APK and force the service to avoid waiting 4 hours:

```
$ adb install new_badnews.apk
$ adb shell am startservice ru.blogspot.playsib.savageknife/com.
      mobidisplay.advertsv1.AdvService -ez update 1
```

3.2 SimpleLocker, a ransomware

Simplelocker [13] is a ransomware discovered in 2014. It encrypts user's multimedia files stored in the SD card. The original files are deleted and the malware asks a ransom to decrypt the files. Our sample displays instructions in Russian. Simplelocker communicates with a server hidden behind a Tor network to receive orders, for example the payment confirmation.

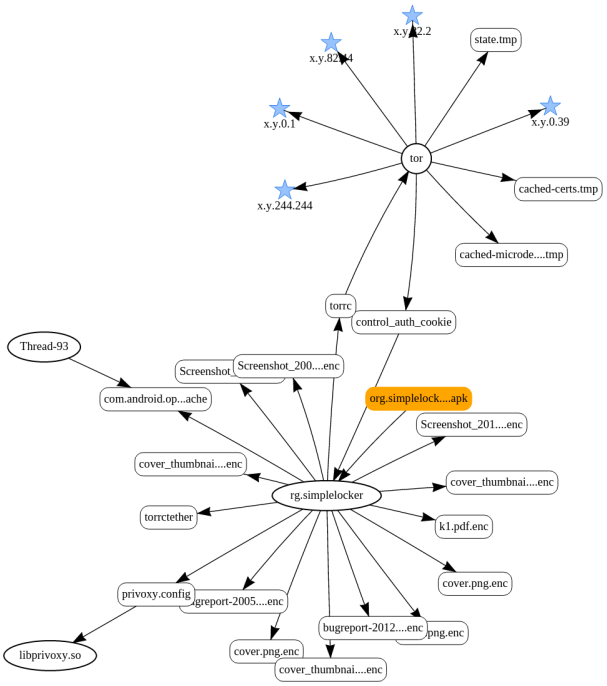

Figure 1: Information flows induced by an execution of SimpleLocker

Simplelocker relies on the execution of three main independent processes. First, *rg.simplelocker* runs the graphical interface, the main service and the different repetitive tasks. Second, *libprivoxy.so* and *tor* are two processes that give access to the Tor network.

Stage 1: Malicious code execution. SimpleLocker waits for the `BOOT_COMPLETED` intent. When it occurs, it starts a service located in the *MainService* class. Starting the main activity with the launcher also starts the service. The service takes a *WakeLock* on the phone in order to get the device running the malware even if the screen goes off. Then, it schedules two repetitive task executors (*MainService$3* and *MainService$4*) and launches a new thread (*MainService$5*). All these jobs are executed in the main process *rg.simplelocker*.

Stage 2: Communication with a server via Tor. A task executor *MainService$3*, launched every 180 seconds, sends an intent *TOR_SERVICE* to start the *TorService* class. If Tor is already up, the *TorSender* class is called to send the IMEI of the phone using the service. The *TorService* class is a huge class that setups executables: it copies and gives executable permission to the files *libprivoxy.so* and *libtor.so* that come from the APK. The *libprivoxy.so* process is executed calling from the *MainService class* as shown below:

```
final String[] array = { String.valueOf(this.filePrivoxy.
      getAbsolutePath()) + " " + new File(this.appBinHome, "
      privoxy.config").getAbsolutePath() + " &" };
TorServiceUtils.doShellCommand(array, sb, false, false);
```

The *libprivoxy.so* process listens for `HTTP` requests on the port 9050. It is an `HTTP` proxy that filters and cleans the request generated and received by the *tor* client.

Stage 3: User's data encryption. In the *MainService$5* thread, the malware encrypts all the multimedia files and deletes the original ones:

```
for (final String s : this.filesToEncrypt) {
  aesCrypt.encrypt(s, String.valueOf(s) + ".enc");
  new File(s).delete(); }
```

The used algorithm is AES in CBC mode with PKCS#7 padding. The encryption key is a constant in the code: we were able to generate a modified version of this malware where we have forced the decryption of the files.

The repetitive task *MainService$4*, checks in the *SharedPreferences* the value *DISABLE_LOCKER*: if set, the malware knows that the victim has paid. If not, it restarts the *Main* activity that displays a fullscreen Russian message informing the user that its files have been encrypted and asking for a ransom.

Triggering Condition. To trigger the malware, launch the application or reboot the device.

Information flow observations. Figure 1 shows that SimpleLocker is constituted of four independent processes (ellipses). The main process named *rg.simplelocker* writes the encrypted version of the multimedia files (`*.enc`). The process named *tor* is the process that communicates through the Tor network, using five sockets (stars). Four of them are nodes of the Tor circuit used to reach the server and the fifth is the interface used to send and receive messages. The *libprivoxy.so* process is the `HTTP` proxy used in combination with Tor.

3.3 DroidKungFu1, a remote admin tool

DroidKungFu1 is a malware discovered in the middle of 2011 that is able to install an application without any notification to the user. We have included this malware in our dataset because it is a well known malware that presents interesting features. We do not give a lot of details about it and we refer the reader to [10, 3].

The malicious code of the malware is included into the package *com.google.ssearch* that contains four classes. The most important class is *SearchService.class*. The malware also comes with 4 noteworthy assets : *gjsvro*, an encrypted version of the *udev* exploit, *ratc*, an encrypted version of the exploit *Rage Against The Cage*, *legacy*, an

APK file that contains a fake Google Search application, and *killall*, a *ratc* wrapper.

Stage 1: Setup of a countdown. DroidKungFu waits the `BOOT_COMPLETED` intent to start the service *SearchService*. When the service starts for the first time, it writes the current time in an XML file `stimestamps.xml` and stops itself. Every time *SearchService* is restarted, it checks if the elapsed time between the time of the restart and the time of `stimestamps.xml` exceeds four hours.

Stage 2: Installation of a fake Google Search app. When the period of four hours has expired, the malware collects sensitive information about the device (the IMEI, the device model, the phone number, the SDK version, memory size, network information) and tries to use the *Exploid* or *RATC* exploits. If it fails, it tries to use the *su* binary to become root. Then it extracts an APK from the asset *legacy*. This APK file is placed into the directory `/system/app` and further detected by `system_server` as a new application to be installed.

Stage 3: Executing the C&C server commands. Then, the malware or the fake Google Search app can receive commands from a remote server. This way, if the originating infected app is removed, the malware can still be able to receive commands through the fake Google Search app. The commands can be: install or delete any package, start an application or open a web page.

Triggering Condition. To trigger this malware, install the application, launch it once and reboot the phone. Then you need to execute the following command:

```
adb pull /data/data/com.allen.mp/shared_prefs/sstimestamp.
    xml
```

Modify the value *start* to 1 and push back the file in the phone. After that, just reboot the phone again.

3.4 MobiDash, an adware

MobiDash [6] is an adware discovered in January 2015. Hidden behind a functional card game, it displays unwanted ads each time the user unlocks the screen. To evade dynamic analysis tools, the malware waits several days before executing its malicious code. For that purpose the malware uses three internal states, namely *None*, *Waiting* and *WaitingCompleted*. The default state is *None*. The malware switches from *None* to *Waiting* when rebooted and reaches the state *WaitingCompleted* after a fixed countdown. Finally, it starts to display ads.

Stage 1: Bootstrapping the configuration. When the application is launched for the first time, the activity *com.cardgame.durak.activities.ActivityStart* is created and it calls the *InitAds()* function from the *MyAdActivity* class. This triggers a bootstrap procedure in which the file `res/raw/ads_settings.json` is read. This file contains information about the malware configuration, and in particular, contains the server to be contacted and the time to wait before triggering (called *OverappStartDelaySeconds*). In our sample, the server is `http://xxx.mads.bz`[6] and the delay is 24 hours. All these parameters are then saved in the *SharedPreferences* and the malware has reached the state *None*.

Stage 2: From state *None* to *Waiting*. Once the device is rebooted, the `BOOT_COMPLETED` intent is received by the *DisplayCheckRebootReceiver* and it triggers the *ping()* function from the *AdsOverappRunner* class. This function checks the internal state of the malware and executes a specific function for each case.

```
final AdsOverappRunner.State state = getState(context);
switch (
$SWITCH_TABLE$mobi$dash$overapp$AdsOverappRunner$State
     ()[state.ordinal()]) {
       case 1: {
              startWait(context);
              break;
       }
       case 2: {
              checkForCompleted(context);
              break;
       }
       case 3: {
              startAds(context);
              break;
       }
}
```

If the state is *None*, it calls the *startWait()* function which changes the internal state into *Waiting*, saves the current time in the *SharedPreferences* and setups two alarms. The first (resp. second) alarm is used to retrigger the *DisplayCheckRebootReceiver* every 15 minutes (resp. 24 hours).

```
protected static void startWait(final Context context) {
       setState(context, AdsOverappRunner.State.Waiting);
       setWaitStartTime(context, System.currentTimeMillis());
       DisplayCheckRebootReceiver.setupPingAlarms(context);
       DisplayCheckRebootReceiver.setupPingAlarmOne(context
            , (long)(AdsExtras.getOverappStartDelaySeconds
            ()*1000+1000)); }
```

Stage 3: From state *Waiting* to *WaitingCompleted*. The next call to *ping()* (with the *Waiting* state) will execute the *checkForCompleted()* function. This function checks if the delay has expired, changes the state to *WaitingCompleted* and calls the *startAds()* function. *startAds()* starts the service *DisplayCheckService* that request ads to the server and display them. Additionally,

the service sets up an alarm, as done in *startWait()*, in order to restart itself every 15 minutes. It also dynamically registers two receivers:

```
protected void setupUserPresent() {
       this.registerReceiver(this.screenOffReceiver, new
            IntentFilter("android.intent.action.SCREEN_OFF")
            );
       this.registerReceiver(this.userPresentReceiver, new
            IntentFilter("android.intent.action.
            USER_PRESENT")); }
```

The first receiver requests new ads each time the screen turns off by calling the *requestAds()* function. The second receiver displays an ad each time the user unlocks the screen by calling the *showLink()* function.

Additional features. When reversing the malware, we observed that the class *HomepageInjector* changes the browser homepage and the class *AdsShortcutUtils* installs launcher shortcuts. During our observations, none of these features have been activated.

We also observed that our malware sample contains a lot of different lawful Advertising Service SDK: *AdBuddiz*, *AdMob*, *Flurry*, *MoPub*, *Chartboost*, *PlayHaven*, *TapIt* and *Moarbile*. Nevertheless, the malware main activity (*ActivityMain$11*) only uses *AdBuddiz*, *AdMob* and *Chartboost*. To finish, log files about all the downloaded malicious ads are stored in the folder `data/data/com.cardgame.durak/files/mobi.dash.history/active/`. These logs contain information such as the requests to the server.

Triggering Condition. First, the application must be launched a first time and the device must be rebooted in order to reach the state *WaitingCompleted*. Then, by setting *waitStartTime* to 0 in the XML file of the directory `/data/data/com.cardgame.durak/shared_prefs/com.cardgame.durak_preferences.xml` and rebooting again, the malicious code is triggered. The smartphone must be rebooted promptly after modifying the file, for example by pushing it with adb, in order to avoid the malware to overwrite the modification.

Information flow observations. We give in Figure 2 the full graph of MobiDash as an example of a malware that generates a lot of system events. The main process `cardgame.durak` reads the file `ads_settings.json` to configure itself and connects to a large amount of IP addresses. Some of those IP are contacted by the originating game itself to retrieve fair ads and most of them are contacted by the malware to download malicious ads. The IP addresses shared between `cardgame.durak` and `android.browser` are connections opened when aggressive ads are displayed in fullscreen in a webview. We notice that the malware saves its history in a local directory, producing a lot of log files.

3.5 SaveMe, a spyware

SaveMe [11] is a spyware discovered in January 2015. It presents itself as a standalone application that is supposed to backup contacts and SMS messages. SaveMe seems to be a variant of another malware known as *SocialPath* [14]. The application has been available on *Google Play* before being removed.

Stage 1: Sensitive data recovery. When the application is launched, it asks to the user his name and phone number and saves these inputs in its local database *user_info4*. In background, the activity collects the device's MAC address, network operator name and ISO country code. Those information are then all sent to a master server, located at `http://xxxxmarketing.com`[7] (no longer available).

The visible part of the application offers features such as: add or delete a contact, save or restore your phonebook, save all your SMS messages and write a SOS message that will be sent to all your contacts in case your phone has been stolen. If you choose to save your messages, the application will save all the content of `content://sms/inbox` and `content://sms/sent` in its local database *user_info* and send it to the server.

Stage 2: Execute the master commands. In parallel, when the application is launched, a service named *CHECKUPD* is started (it also starts each time the device is rebooted). This service is used as a handshake between the device and the server. It executes three *AsyncTask* namely *sendmyinfos()*, *sendmystatus()* and *senddata()* for dialoging with the server. After those exchanges, the main service *GTSTSR* is executed. The purpose of this service is to contact the server in order to get commands to be executed. Depending on the answer given by the server, the service can perform different actions as detailed below.

First, it can send a text message to any number given by the server. We believe that this can be used for premium services as stated in [14].

```
if (GTSTSR.Mac.equals(this.address) && GTSTSR.
     Send_ESms.equals("SESHB")){
new update().var(this.address,"","SESFK","","","","","");
SmsManager.getDefault().sendTextMessage(GTSTSR.
     EXT_SMS, null, GTSTSR.SMS, null, null);
return; }
```

The service can also make a call by starting a service named *RC*. This service displays a *WebView* on the screen, probably to hide the call and makes a call to a potentially premium number given by the server [14].

```
Intent localIntent = new Intent("android.intent.action.CALL");
localIntent.setData(Uri.parse("tel:" + EXT_CALL));
intent.addFlags(268435456); intent.addFlags(4);
this.startActivity(intent);
```

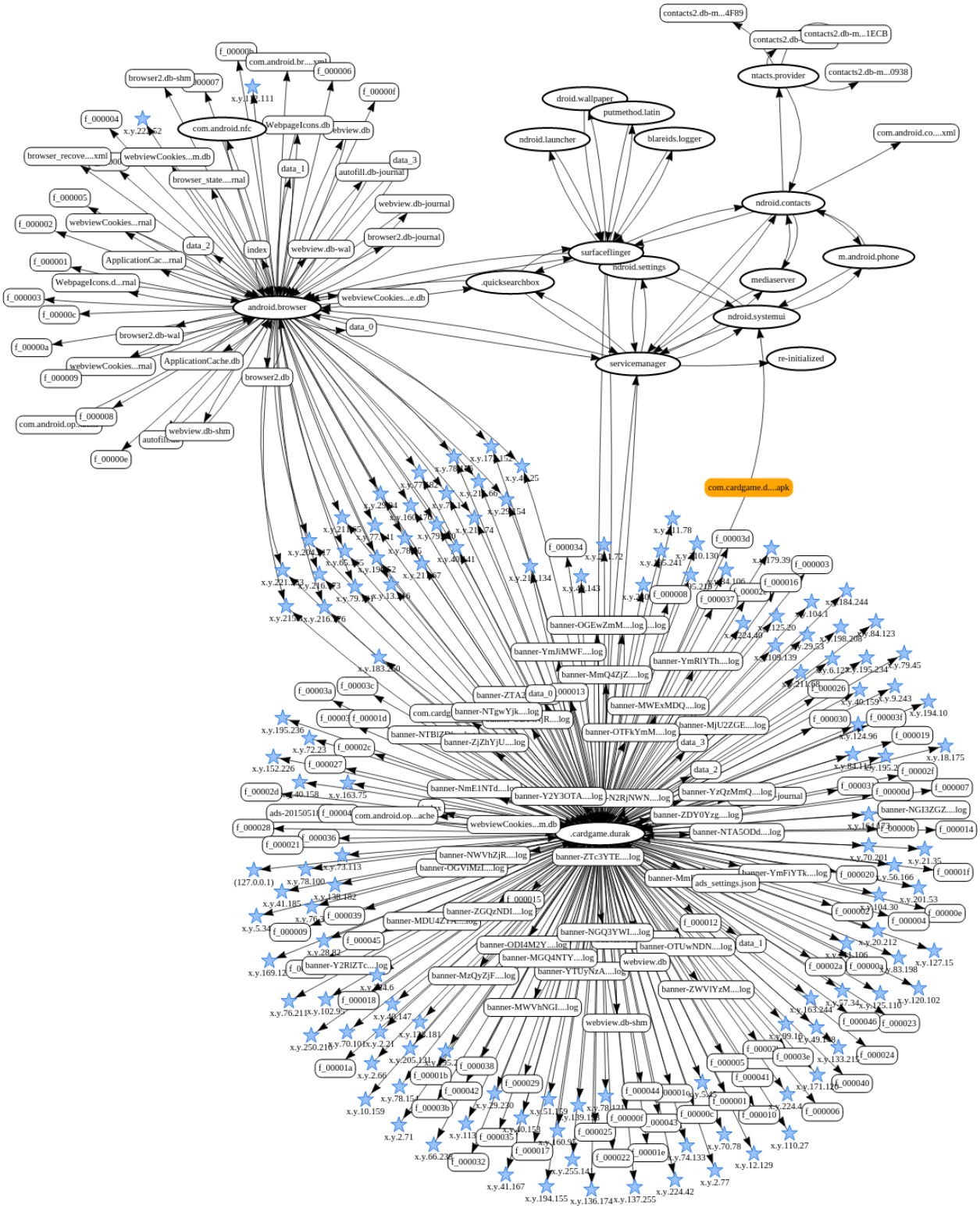

Figure 2: Information flows induced by an execution of MobiDash

After few moments, the service ends the call, removes the *WebView* and deletes the call in the call log by calling the function *DeleteNumFromCallLog()*.

```
final Uri parse = Uri.parse("content://call_log/calls");
contentResolver.delete(parse, "number=?", new String[]{s});
```

GTSTSR can also start a service named *CO* which will automatically fetch all the contacts of the victim and send them to the server. The main difference compared with the official feature of the application (except that there is no need to click on a button) is that *CO* will also steal contacts stored in the SIM card by reading `content://icc/adn`. Contacts are then stored in the database *user_info* before being sent.

The last feature provided by *GTSTSR* is the sending of text messages to victim's contacts by starting the service *SCHKMS*. The service checks the database *user_info*, picks one contact and sends him a message. This feature is used for spreading the malware via SMS containing a link [14]. Of course, the service deletes the SMS from the logs in order to hide it to the victim.

To finish with this malware, we observed a piece of code in the activity *pack* which allows the app to remove its icon from the launcher, in order to hide itself. This way, the victim may forget to uninstall the application. Nevertheless, this activity is never used in this sample.

```
this.getPackageManager().setComponentEnabledSetting(this.
    getComponentName(),
    COMPONENT_ENABLED_STATE_DISABLED,
    DONT_KILL_APP);
```

Triggering Condition. To trigger this malware it is sufficient to use the application icon or to reboot the device. Internet must be enabled for the malware to start.

3.6 WipeLocker, a blocker and data eraser

WipeLocker [5] is a malware discovered in September 2014. It blocks some social apps with a fullscreen hacking message and wipes off the SD card. It also sends SMS messages to victim's contacts. It might be an app for helping the sell of antivirus.

The malware presents itself as a fake *Angry Bird Transformers* game. Once the application is launched, the main activity performs three actions.

Stage 1: Starting the malicious service. The application first starts the service *IntentServiceClass* that can also be triggered by the `BOOT_COMPLETED` event. This service schedules the execution of *MyServices.getTopActivity()* every $0.5s$ and *MyServices.Async_sendSMS()* every $5s$. *getTopActivity()* checks the current foreground activity: if it is a social app like *Facebook*, *Hangouts* or *WhatsApp*, it displays a fullscreen image "*Obey or Be Hacked*", making impossible to use those apps. *MyServices.Async_sendSMS()* is an AsyncTask that sends a text message every $5s$ to all the victim's contacts: "*HEY!!! <contact_name> Elite has hacked you.Obey or be hacked*".

Stage 2: Activating the device admin features. The second action of the malware is to ask the user to activate the device administration features of the app [9]. If the user declines, the app will ask again, over and over, until the user accepts to do so. Administration features allow an application to perform sensitive operations such as wiping the device content or enforcing a password security policy. The file `res/xml/device_admin_sample.xml` declares the operations the application intends to handle. The content of this file is however empty, which means that the application will not handle any sensitive operations: the purpose of this stage is to make the app much harder to uninstall because device administrators cannot be uninstalled like normal apps. If the user accepts, the app closes itself and remove its icon from the launcher.

Stage 3: Wiping off the SD card. The last action performed by the malware is the deletion of all the files and directories of the external storage. Even if the user declined the device administration features, the function *wipeMemoryCard()* is called. This function uses *Environment.getExternalStorageDirectory()* to get the path of the external storage, and then calls *File.listFiles()* for iterating on files and deleting each of them.

Stage 4: Intercepting SMS. A last feature that comes with the malware is the interception of incoming SMS. It is simply a receiver named *SMSReceiver* that is triggered by the `SMS_RECEIVED` intent. When an SMS is received, the malware automatically answers to the sender with the message "*Elite has hacked you.Obey or be hacked*". The victim is not notified by the system about any incoming SMS because the receiver has a high priority (2147483647 in the manifest) and calls *abortBroadcast()* just after reading the message.

Triggering Condition. The icon launcher triggers all the features. A reboot of the device triggers the service.

3.7 Cajino, a spyware

Cajino is a spyware discovered in March 2015. Its particularity is to receive commands via *Baidu Cloud Push* messages. In addition to alternative markets, samples were downloadable on the *Google Play* store with more than 50.000 downloads.

Stage 1: Registration. The application must be launched at least one time. When it occurs, in the *onCreate()* function, a registration procedure of the *Baidu* API is executed in order to make the phone able to receive Push messages from the remote server.

```
if(!Utils.hasBind(this.getApplicationContext())){
        PushManager.startWork(this.getApplicationContext(),
        0, Utils.getMetaValue((Context)this, "api_key")); }
```

At the same time, the *MainActivity* displays an empty *WebView* and a dialog box pops up asking for an update with a "Yes" or "No" choice, with no code behind.

Stage 2: Receiving Push messages. The malware has a receiver named *PushMessageReceiver*. It can react to these intents broadcasted by *Baidu* services:

```
com.baidu.android.pushservice.action.MESSAGE
com.baidu.android.pushservice.action.RECEIVE
com.baidu.android.pushservice.action.notification.CLICK
```

When a Push message is received, *PushMessageReceiver* starts *BaiduUtils.getFile()* which checks if the device is concerned by the incoming message, and if so, starts *BaiduUtils.getIt()* to execute the right command. The commands are designed to: steal the contacts, steal the call logs, steal all SMS (inbox and sent), get the last known location of the device, steal sensitive data (IMEI, IMSI, phone number), list all data stored on the external storage. For each of these features, the malware first stores the results in files written into `/sdcard/DCIM/Camera/` before uploading them to the remote server. The malware can also send SMS to any phone number given by the server, upload to the server or delete any file stored on the external storage.

In some other versions, e.g. *ca.ji.no.method2*, more features are available. For example it can record the microphone with a *MediaRecorder* during a period of time given by the server:

```
BaiduUtils.recorder.prepare(); BaiduUtils.recorder.start();
Thread.sleep(int1 * 1000);
BaiduUtils.recorder.stop(); BaiduUtils.recorder.release();
```

It can also download an APK file into the directory `/sdcard/update/` and install it on the device:

```
private static void installApk(final Context context, String
      str) {
str = Environment.getExternalStorageDirectory()
        + "/update/update.apk";
final Intent intent = new Intent("android.intent.action.VIEW");
intent.addFlags(268435456);
intent.setDataAndType(Uri.fromFile(new File(str)),
        "application/vnd.android.package-archive");
context.startActivity(intent); }
```

The last feature of *Cajino* is a classical call to a number given by the server, not hidden from the user. That makes a total of 12 distinct features the malware can perform.

Triggering Condition. Launch the app to trigger the registration, then you need to wait for a Push message from the remote server. If you want to force the execution of a command, for example for listing the files of `/sdcard/`, send an intent with adb:

```
adb shell am broadcast -a com.baidu.android.pushservice.action
      .MESSAGE --es message_string "all list_file"
```

3.8 Dataset summary and usage

Table 2 gives an overview of the studied malware. For each of them, we recall their protection against dynamic analysis and give the main actions for defeating these protections. These remediation techniques will support the reproducibility of future research experiments.

We have used our dataset to evaluate the performances of GroddDroid [1], a tool for triggering malicious behaviors that targets suspicious methods. On four of them, the targeted methods were automatically triggered. On MobiDash, a method with benign code were targeted (false positive) and on SimpleLocker, GroddDroid had a crash. This example shows that documented dataset helps to measure if a proposed method works fine. Of course, for larger datasets, an other approach should be used to compute the false positive/negative results, but the use of Kharon dataset gives an opportunity to carefully check if a tool works as expected.

4 Conclusion

In this article, we have proposed to initiate the construction of a dataset of seven Android malware that illustrate as much as possible existing malware behaviors. These malware are recent, from 2011 to 2015. For all of them we detailed their expected behavior, isolated the malicious code and we observed their actions in a controlled smartphone. All these materials can be found online on the Kharon website.

An important result of this study is that these malware present a pool of techniques to hide themselves from dynamic analyzers. Thus, we explain how to trigger their malicious code in order to increase the reproducibility of research experiments that need malware execution.

We continue to supply the dataset and additional descriptions of malware can be read. We also propose to other actors of the community to enlarge this dataset. For that purpose, we encourage researchers to gather their experience by signaling us their own documentation about reversed android malware. We will be pleased to integrate any contribution. This way, we hope that this effort will bring new inputs for the research community and will become a reference dataset for precise and reproducible malware analysis.

Table 2: Malware dataset summary

Type	Name	Discovery	Protection against dynamic Analysis → *Remediation*	Details for reproducibility
Remote Admin Tool	Badnews	2013	Obeys to a remote server and delays the attack → *Modify the apk* → *Build a fake server*	Section 3.1
Ransomware	SimpleLocker	2014	Waits the reboot of the device → *send a BOOT_COMPLETED intent*	Section 3.2
Remote Admin Tool	DroidKungFu	2011	Delayed Attack → *Modify the value* `start` *to* `1` *in sstimestamp.xml*	Section 3.3
Adware	MobiDash	2015	Delayed Attack → *Launch the infected application, reboot the device and modify com.cardgame.durak_preferences.xml*	Section 3.4
Spyware	SaveMe	2015	Verifies the Internet access → *Enable Internet access and launch the application*	Section 3.5
Phone Blocker + Data Eraser	WipeLocker	2014	Delayed Attack → *Press the icon launcher and reboot the device*	Section 3.6
Spyware	Cajino	2015	Obeys to a remote server → *Simulate the remote server by sending an intent*	Section 3.7

Future works deal with comparing these seven malware with larger datasets in order to build automatic classification techniques. Moreover, for advanced malware that implement sophisticated protections such as obfuscation or ciphering, new investigations should be designed in order to link the static analysis of the code with dynamic analysis.

5 Most important malicious functions

In the following, we give the most 5 most important functions of each malware. It may help researchers to check that their experiment successfully executes the malicious code.

```
# Badnews

com.mobidisplay.advertsv1.AdvService.fillPostData()
com.mobidisplay.advertsv1.AdvService.onStartCommand(final
      Intent intent, final int n, final int n2)
com.mobidisplay.advertsv1.AdvService.startUpdater()
com.mobidisplay.advertsv1.AdvService.sendRequest(String string
      )
com.mobidisplay.advertsv1.AReceiver.onReceive(Context context
      , Intent intent)

# SimpleLocker

org.simplelocker.MainService.onCreate()
org.simplelocker.MainService$4.run()
org.simplelocker.TorSender.sendCheck(final Context context)
org.simplelocker.FilesEncryptor.encrypt()
org.simplelocker.AesCrypt.AesCrypt(final String s)

# DroidKungFu

com.google.ssearch.SearchService.onCreate()
com.google.ssearch.SearchService.updateInfo()
com.google.ssearch.SearchService.cpLegacyRes()
com.google.ssearch.Utils.decrypt(final byte[] input)
com.google.ssearch.Utils$PkgManager.deleteApp(final Context
      context, final String str)

# MobiDash

myutils/activity/MyAdActivity.InitAds(int n, ChartboostDelegate
      chartboostDelegate, int n2)
mobi/dash/overapp/AdsOverappRunners.ping(final Context
      context)
mobi/dash/overapp/AdsOverappRunners.startWait(final Context
      context)
mobi/dash/overapp/AdsOverappRunners.checkForCompleted(
      final Context context)
mobi/dash/overapp/DisplayCheckService.setupUserPresent()

# SaveMe

com.savemebeta.GTSTSR.CHECK()
com.savemebeta.RC.callnow()
com.savemebeta.LogUtility.DeleteNumFromCallLog(final
      ContentResolver contentResolver, final String s)
com.savemebeta.CO.allSIMContact()
com.savemebeta.SCHKMS.fetchContacts()

# WipeLocker

com.elite.MainActivity.onCreate(final Bundle bundle)
com.elite.MainActivity.wipeMemoryCard()
com.elite.MyServices.Async_sendSMS.doInBackground(Void ...
      arrvoid)
com.elite.MyServices.getTopActivity(final Context context)
com.elite.MainActivity.HideAppFromLauncher(final Context
      context)

# Cajino

ca.ji.no.method3.MainActivity.onCreate(Bundle bundle)
ca.ji.no.method3.BaiduUtils.getIt(final String s, final
      Context context)
ca.ji.no.method3.BaiduUtils.getLocation(final Context context
      , final String s)
ca.ji.no.method3.BaiduUtils.sendSMS(final String s, final
      String s2)
ca.ji.no.method2.BaiduUtils.installApk(final Context context,
      String string)
```

6 Availability

All malware descriptions and graphs can be accessed online at:

`http://kharon.gforge.inria.fr/dataset`

7 Acknowledgments

This work has received a French government support granted to the COMIN Labs excellence laboratory and managed by the National Research Agency in the "Investing for the Future" program under reference ANR-10-LABX-07-01.

We would like to thank the research engineers and engineering students of the Cyber Security Master of CentraleSupélec and Telecom Bretagne, who participated to the reverse engineering of the dataset. Our special thanks go to Radoniaina Andriatsimandefitra, Béatrice Bannier, Sylvain Bale, Etienne Charron, Loïc Cloatre, Marc Menu, Guillaume Savy.

References

[1] ABRAHAM, A., ANDRIATSIMANDEFITRA, R., BRUNELAT, A., LALANDE, J.-F., AND VIET TRIEM TONG, V. GroddDroid: a Gorilla for Triggering Malicious Behaviors. In *10th International Conference on Malicious and Unwanted Software* (Fajardo, Puerto Rico, oct 2015), IEEE Computer Society, pp. 119–127.

[2] ANDRIATSIMANDEFITRA, R., AND VIET TRIEM TONG, V. Capturing Android Malware Behaviour using System Flow Graph. In *8th International Conference on Network and System Security* (Xi'an, China, Oct. 2014), M. H. Au, B. Carminati, and C.-C. J. Kuo, Eds., Springer International Publishing, pp. 534–541.

[3] ARSENE, L. An android malware analysis: Droidkungfu, Nov. 2012. `http://www.hotforsecurity.com/blog/an-android-malware-analysis-droidkungfu-4474.html`.

[4] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND MCDANIEL, P. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, UK, jun 2014), vol. 49, ACM Press, pp. 259–269.

[5] CHRYSAIDOS, N. Android WipeLocker - Obey or be hacked, Sept. 2014. `http://www.virqdroid.com/2014/09/android-wipelocker-obey-or-be-hacked.html`.

[6] CHYTRY, F. Apps on google play pose as games and infect millions of users with adware, Feb. 2015. `https://blog.avast.com/2015/02/03/apps-on-google-play-pose-as-games-and-infect-millions-of-users-with-adware/`.

[7] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. 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, Oct. 2010), USENIX Association, pp. 393–407.

[8] GOOGLE. Android security 2014 year in review. `https://static.googleusercontent.com/media/source.android.com/en//devices/tech/security/reports/Google_Android_Security_2014_Report_Final.pdf`.

[9] GOOGLE. Device administration. `https://developer.android.com/guide/topics/admin/device-admin.html`.

[10] JIANG, X. Security alert: New sophisticated android malware droidkungfu found in alternative chinese app markets, May 2011. `http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu.html`.

[11] LINDEN, J. The privacy tool that wasn't: SocialPath malware pretends to protect your data, then steals it, Jan. 2015. `https://blog.lookout.com/blog/2015/01/06/socialpath/`.

[12] LINDORFER, M., NEUGSCHWANDTNER, M., WEICHSELBAUM, L., FRATANTONIO, Y., VAN DER VEEN, V., AND PLATZER, C. Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security* (Wroclaw, Poland, Sept. 2014).

[13] LIPOVSKY, R. ESET analyzes first android file-encrypting, TOR-enabled ransomware, June 2014. `http://www.welivesecurity.com/2014/06/04/simplocker/`.

[14] NEMCOK, M. Warning: Mobile privacy tools "socialpath" and "save me" are malware, Jan. 2015. `http://blog.lifars.com/2015/01/11/warning-mobile-privacy-tools-socialpath-and-save-me-are-malware/`.

[15] PARKOUR, M. Contagio mobile, 2012. `http://contagiominidump.blogspot.fr/`.

[16] ROGERS, M. The bearer of BadNews, Mar. 2013. `https://blog.lookout.com/blog/2013/04/19/the-bearer-of-badnews-malware-google-play/`.

[17] ROSSOW, C., DIETRICH, C. J., GRIER, C., KREIBICH, C., PAXSON, V., POHLMANN, N., BOS, H., AND VAN STEEN, M. Prudent practices for designing malware experiments: Status quo and outlook. In *IEEE Symposium on Security and Privacy* (San Francisco Bay Area, CA, USA, may 2012), IEEE Computer Society, pp. 65–79.

[18] STEFANKO, L. Remote administration trojan using baidu cloud push service, Mar. 2015. `http://b0n1.blogspot.fr/2015/03/remote-administration-trojan-using.html`.

[19] ZHOU, Y., AND JIANG, X. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy* (San Francisco Bay Area, CA, USA, may 2012), IEEE Computer Society, pp. 95–109.

Notes

[1] `https://github.com/secure-software-engineering/DroidBench`

[2] We warn the readers that these samples have to be used for research purpose only. We also advise to carefully check the SHA256 hash of the studied malware samples and to manipulate them in a sandboxed environment. In particular, the manipulation of these malware impose to follow safety rules of your Institutional Review Boards.

[3] `https://play.google.com/store/apps/details?id=com.noshufou.android.su`

[4] `https://www.blare-ids.org`

[5] We intentionally anonymized this URL

[6] We intentionally anonymized this URL

[7] We intentionally anonymized the URL