Mobile-Sandbox: Having a Deeper Look into Android Applications

Michael Spreitzenbarth, Felix Freiling
Friedrich-Alexander-University
Erlangen, Germany
michael.spreitzenbarth, felix.freiling@cs.fau.de

Florian Echtler, Thomas Schreck
Siemens CERT
Munich, Germany
florian.echtler, t.schreck@siemens.com

Johannes Hoffmann
Ruhr-University Bochum
Bochum, Germany
johannes.hoffmann@rub.de

ABSTRACT

Smartphones in general and Android in particular are increasingly shifting into the focus of cybercriminals. For understanding the threat to security and privacy it is important for security researchers to analyze malicious software written for these systems. The exploding number of Android malware calls for automation in the analysis. In this paper, we present *Mobile-Sandbox*, a system designed to automatically analyze Android applications in two novel ways: (1) it combines static and dynamic analysis, i.e., results of static analysis are used to guide dynamic analysis and extend coverage of executed code, and (2) it uses specific techniques to log calls to native (i.e., "non-Java") APIs. We evaluated the system on more than 36,000 applications from Asian third-party mobile markets and found that 24% of all applications actually use native calls in their code.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

Keywords

Android, Malware, Application analysis

1. INTRODUCTION

1.1 Android (Malware) on the Rise

In recent years smartphone sales tremendously increased. This explosive growth has drawn the attention of criminals who try to attract the user to install malicious software on the device. Google's smartphone platform Android is the most popular operating system and recently overtook Symbian- and iOS-based installations. Most probably, this growth stems from the openness of the platform which allows a user to install arbitrary software.

But attackers are misusing this openness to spread malicious applications through common Android application markets. In previous work [22] we analyzed about 6,100 malicious applications and clustered them into 53 malware families with the help of the *VirusTotal API* [15]. Nearly 57% of our analyzed malware families tried to steal personal information from the smartphone like address book entries, the IMEI or GPS coordinates. Additionally, sending SMS messages rates with about 45%. Most common was sending these messages to premium rated numbers to make money immediately. The last main feature which was implemented in nearly 20% of the malware families is the ability to connect to a remote server in order to receive and execute commands. Another detailed and well-readable overview of all these existent malware families is provided by Zhou et al. [27].

1.2 The Need for Automated Analysis

Given the enormous growth of Android malware, security researchers and vendors must analyze more and more applications (apps) in a given period of time to understand the purpose of the software and to develop countermeasures. Until recently, analysis was done manually by using tools like decompilers and debuggers. This process is very time consuming and error-prone depending on the skill set of the analyst. Therefore, tools for automatic analysis of apps were developed.

The classical approach to automated analysis is *static* analysis. Static analysis investigates software properties that can be investigated by inspecting the downloaded app and its source code only. Signature-based detection of apps, the common approach by anti-virus technologies, is an example of static analysis. In practice, malware uses obfuscation techniques to make static analysis harder. A particular form of obfuscation used by Android apps is to hide system activities by calling functions outside the Dalvik/Java runtime library, i.e., in *native* libraries written in C/C++ or other programming languages.

In contrast to static analysis, *dynamic* analysis does not inspect the source code but rather executes it within a controlled environment, often called sandbox. By monitoring and logging every relevant operation of the execution, a report is automatically generated for each analysis. Dynamic analysis can combat obfuscation techniques rather well but can be circumvented by runtime detection methods. Therefore, it usually makes sense to combine static and dynamic analysis which can be done in many different ways.

SAC'13 March 18-22, 2013, Coimbra, Portugal.

1.3 Existing Android Analysis Systems

Similar to the development in the desktop PC world, the early systems for analysis of Android apps used a static approach. A typical system for this approach was proposed by Schmidt et al. [20]. They attempt to extract the function calls from an Android application (using the *readelf* utility) and compare the resulting list with the data of known malware. Another example for the static approach is *Androguard* by Desnos et al. [5, 6] which decompiles the application and applies signature based malware detection. This system is completely open source.

In response to static analysis systems in the desktop PC world, malware authors developed various obfuscation techniques that have have proven their effectiveness against static analysis [17, 24]. This is also an emerging trend in Android applications and it is clear that static analysis alone cannot ensure complete analysis coverage anymore. Therefore, researchers have begun to develop systems for dynamic analysis of Andoid apps.

One of the first such systems is *TaintDroid* by Enck et al. [7]. It is an efficient and dynamic taint tracking system which provides realtime analysis by leveraging Android's execution environment. This system was complemented with a fully automated user emulation and reporting system by Lantz [16] and is available under the name *Droidbox*. Droidbox is an effective tool to analyze Android apps, however, it lacks support to track native API calls. In fact, we are unaware of any system that supports native API call tracking during dynamic analysis to date. Both tools are open source and publicly available.

Another interesting system using dynamic analysis is *pBMDS* by Xie et al. [26]. It uses machine learning to create user and system profiles for a specified behavior. Afterwards, it tries to correlate user inputs with system calls by comparing their behavior profiles to detect anomalous application activities. This system was built for Symbian OS and tested with a very small data-set. *Crowdroid*, by Burguera [3] uses a similar approach but with a much wider set of behavior data and with a more advanced monitoring system. CrowDroid uses `strace`, a debugging utility for Linux and some other Unix-like systems, to monitor every system call and the signals it receives. Crowdroid, however, does not consider information from Android's Dalvik VM. The system *AASandbox* of Bläsing et al. [2] was the first system combining static and dynamic analysis in a very basic way for the Android platform. Unfortunately, AASandbox does not seem to be maintained anymore. Another system combining static and dynamic analysis is *DroidRanger* by Zhou et al. [28]. DroidRanger implements a combination of permission-based behavioral footprinting to detect samples of already known malware families and a heuristic-based filtering scheme to detect unknown malicious families. With this approach they were able to detect 32 malicious samples inside the official Android Market in June 2011. Within their dynamic part they use a kernel module to log only system calls used by known Android exploits or malware.

The system which is most similar to our approach is *Andrubis* from the Vienna University of Technology [18]. In their approach they also use Droidbox and TaintDroid for automated analysis but they are limited to applications beneath API level 8 (Android 2.3). In contrast, we are able to analyze applications beneath API level 11. This difference can be very important when you compare the market-share of API level 7 and below (17 percent) to the share of API level 10 and below (75 percent). Additionally, they are not able to track native code.

1.4 Contribution: Mobile-Sandbox

Overall, there are only few analysis systems that combine static and dynamic analysis and none that dynamically monitor both actions within the Dalvik VM and outside it in native libraries. Moreover, many of these systems are not readily available for research or are not maintained anymore. In this paper, we seek to fill this gap by introducing *Mobile-Sandbox*, a system that

1. uses a novel combination of static and dynamic analysis techniques,
2. can track native API calls, and
3. is easily accessible for everyone through a web interface [4].

Within the static analysis part we analyze the application with various modules to get an overview of the application. First, we perform several anti-virus scans using the VirusTotal service [15], secondly, we parse the manifest file, and finally we decompile the application to better identify suspicious code.

Within the dynamic analysis, we execute the application in an emulator and log *every* operation of the application, i.e., we log both the actions executed in the Java Virtual Machine *Dalvik* and actions executed in native libraries which may be bundled with the application. To be best of our knowledge, Mobile-Sandbox is the first analysis framework for the Android platform which has this capability.

For evaluating our system we collected over 136,000 freely available apps from the most important Asian markets and the Google Play-Market. We also collected about 7,500 malicious samples from different malware families. We then used Mobile-Sandbox to automatically analyze 40,000 randomly chosen apps from both sample sets. Within these 40,000 samples our system detected 4,641 malicious applications and additionally 5 suspicious samples which try to hide their malicious action inside native code. This insight clearly indicates that current analysis systems are overlooking important potential threats.

1.5 Roadmap

The remainder of this paper is organized as follows: Section 2 characterizes the current threat landscape in mobile devices especially for malware on the Android platform and gives some background on the Android platform. In Section 3 we illustrate our framework and explain the main ideas behind our static and dynamic analysis. In Section 4 we present the results of our evaluation. We conclude in Section 5.

2. BACKGROUND

2.1 The Android Threat Landscape

Mobile threats can be categorized into two classes: *web-based* and *application-based* threats. Web-based threats on mobile devices are a growing attack vector used by criminals. These threats rely on the enormous usage of mobile browsers and their feature-rich implementations. Modern

web browsers support features like embedded video players or support for video calls. Due to the nature of these features, e.g., parsing huge amounts of external data, the possibility for the existence of exploitable vulnerabilities is high. Attackers are able to trick the user to follow a web link, sent to them via email or social media, and infect the smartphone by exploiting a browser vulnerability.

The other type of mobile threats are application-based threats posed by third-party applications in the mobile markets. To install applications on the smartphones the vendors created so-called *mobile markets* like Apple's "App Store" and Google's "Google Play". On iOS-based devices, software can be obtained from the App Store only. Furthermore, Apple evaluates every software uploaded to the App Store and only adds the app if it passes certain (unknown) security checks. On Android devices the end user is also allowed to install apps from third-party markets. Especially in Asia a lot of these markets emerged. Typically these third-party markets pose a high risk to install malicious applications due to the fact that the market owners do not evaluate the applications.

According to Felt et al. [8], mobile applications pose the following three types of threats – *Malware*, *Personal Spyware* and *Grayware*. This illustrates that the threat landscape for Android is real and relevant.

2.2 Android System Basics

We briefly introduce Android and its relevant parts for this paper in this section. For a thorough introduction we refer to Six [21].

Android is based on Linux and therefore consists of the same core components as usual Linux distributions do. The core components are a (patched) Linux kernel, the Bionic libc and libraries like WebKit, SQLite and OpenGL. The Android runtime environment consists of core libraries which provide most functionality provided by the core Java libraries. It additionally consists of the Dalvik Virtual Machine which is responsible for running Android applications in the operating system. Applications are written in the Java language and each application is executed in its own Dalvik VM. This VM runs dalvik-dex code which is translated from Java bytecode. Dex code is an optimized bytecode suited for mobile devices; the biggest difference is that dex code is register based instead of stack based, as is "traditional" Java bytecode.

One relevant feature of the Dalvik VM for this paper is the ability that applications written in the Java language can additionally access native libraries through the *Native Development Kit* (NDK) which makes use of the *Java Native Interface* (JNI). Developers may move performance critical operations to native libraries (shared objects in the *ELF* format) which are then directly called from running dex code. The native code contained in such libraries runs outside the Dalvik VM directly on the processor of the smartphone or emulator.

3. MOBILE-SANDBOX: ARCHITECTURE AND IMPLEMENTATION

In order to determine whether an app is malicious or not, it needs to be analyzed with great effort. Its attributes as well as the function range need to be documented. Within this section we describe the process of our automated analysis system. The analysis process has been divided into two parts. At first, we discuss the static analysis in Section 3.1. The results of the static analysis are used to guide the following dynamic analysis which is described in Section 3.2. The dynamic analysis automatically executes the apps on a modified Android system with the help of the Android emulator.

3.1 Static analysis

Our static analysis consists of several modules. In order to gain a first impression of the application that should be analyzed, the corresponding hash value is matched with the VirusTotal database. The received detection rate is stored for the report. However, it does not play a vital role within further processing. The result that is delivered by Kaspersky (one part of the VirusTotal scanning engine), is used for the classification into existing malware families.

Afterwards, the application is extracted in order to get access to its components; these are required for further analysis. As a following step, we analyze the Android manifest to get a listing of all required permissions. For this reason we use the tool `aapt` being delivered with the Android SDK [11]. While parsing the manifest we filter the intents as well as the services and receiver for further analysis, too. We also read out the SDK-version; this is another important detail to assure that only applications being compatible to the provided Android system are passed on to dynamic analysis.

Now, the Dalvik byte code that is stored in the `classes.dex` file is converted to `smali` [10]. We can determine and filter the embedded advertising networks from the resulting files and their directory structure as not to dilute the results of the analysis. Afterwards, the complete smali-code is searched for potentially dangerous functions and methods. Here, we take care of calls that can be found frequently and in particular within malware. This includes, for example:

- `sendTextMessage()`: This call is responsible for the sending of SMS messages
- `getPackageInfo()`: With the help of this method, malware often searches for installed AV products
- `getSimCountryIso()`: This call is used to find out in which country the user currently resides. This is important in case of malware to contact the right premium services
- `Ljava/lang/Runtime;->exec()`: Executes the specified command in a separate process. In case of malware, the commands are often 'su' or 'chmod'

Moreover, we look for calls of the available encryption libraries. With this step we try to gain deeper knowledge of the use of encryption and obfuscation within the applications. During the code-review we try to recognize the functions and methods that normally need a permission for their error-free execution. For this reason we refer to data of Felt et al. [9] which we translated from Java to smali. With the help of the gathered list we can now compare if the app is over- or underprivileged. As an ongoing step, it is searched for statically coded URL's with the help of a regular expression.

We filter all implemented timers and broadcasts the app is waiting for, as a preparation for the dynamic analysis. Timers and broadcasts are event triggers for certain code to

be executed in Android. We analyze these mechanisms to either trigger the corresponding events or wait for a specified time period in the ongoing dynamic analysis to improve the coverage of executed code. By this we assure for example that the analysis is not stopped before a timer has expired. This is a common problem in Windows-based dynamic analysis [25].

At the end of the static analysis a XML file is created, containing all data generated during the previous steps.

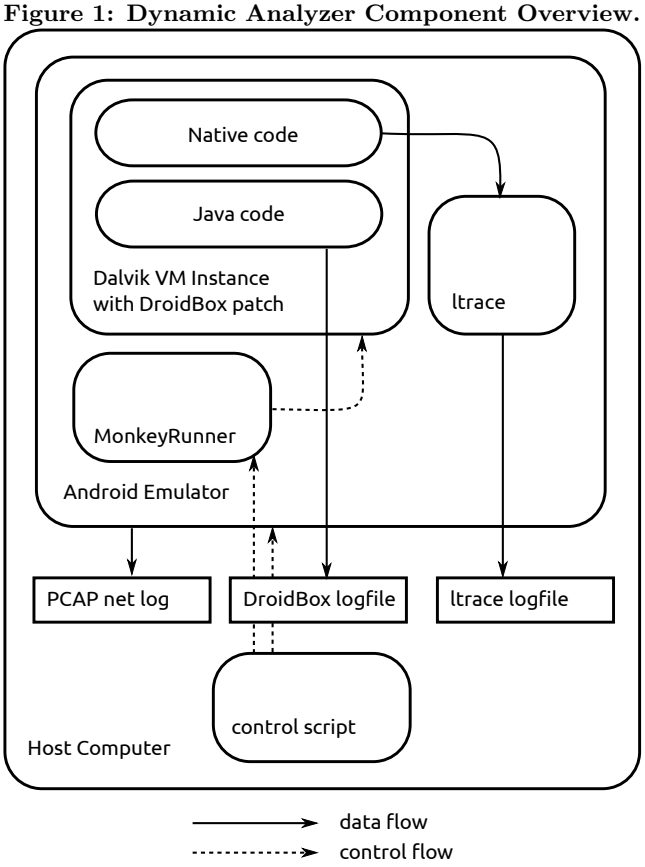
3.2 Dynamic analysis

While certain types of malicious behavior can already be recognized through static analysis, many kinds of malware can only be reliably detected by looking at its runtime behavior.

3.2.1 Building Blocks

To perform such a dynamic analysis of the app in question, we rely on the Android emulator provided by Google [1]. This software simulates a full ARMv7 device with key peripherals such as GSM module and touchscreen, thereby allowing to run unknown apps in a safe environment on a host computer. As an added benefit, the emulator can be reset to its previous state after an app has been tested by simply replacing the system image files with the original ones. Figure 1 gives an overview of the integration of the emulator into the entire Mobile-Sandbox framework.

Figure 1: Dynamic Analyzer Component Overview.



Since the "stock" emulator offers only limited logging capabilities, we have chosen the well-known *TaintDroid/DroidBox* [7, 16] system as basis for our dynamic analyzer. TaintDroid focuses on providing real-time privacy information to a user on a private device, while DroidBox builds on this work by logging all data accessed by the app to the system log, thereby creating a comprehensive picture of the app's runtime activities. This includes data read from and written to files, sent and received over the network, SMS messages sent, and many more.

While TaintDroid supports Android up to version 2.3.4, thereby covering at least 75 % of the device market as of Oct. 2012, DroidBox only supports Android up to version 2.1 which can be considered severely outdated. Therefore, we updated the DroidBox patchset to work with TaintDroid 2.3.4, including some additional enhancements such as support for UDP traffic logging.

3.2.2 Tracking Native Code

However, this setup still has a "blind spot": since both TaintDroid as well as DroidBox are built on the Dalvik virtual machine used by Android to execute dex bytecode, only dex bytecode (in general translated from Java bytecode) can be traced by those tools. Native code executed using the JNI will not be visible. Since the introduction of the Android NDK, it is easily possible to call native code in external libraries. Although many higher-level functions of Android are available only in Java, some lower-level functions such as `socket()`, `connect()`, `read()` or `write()` can be easily called from the standard C library and could be used by malware for communication purposes. While the app would still require the corresponding permissions such as `android.permission.INTERNET` for being allowed to open network sockets, a purely Java-based call tracer would not be able to detect any communication conducted using native calls.

Consequently, the dynamic analyzer should have the ability to trace code included in native shared objects, those included with the app through the NDK as well as those shipped with Android. For this purpose, we have included a modified version of the `ltrace` [23] command, a common Linux debugging utility that intercepts library calls of a monitored application. After the app to be examined has been started, an `ltrace` instance is launched and attached to the Dalvik process running the app in question. All native calls made into dynamically loaded shared objects are then logged to a separate file. To reduce the amount of log data and increase performance, internal functions commonly introduced by the NDK compiler are excluded from logging. Such functions are for example `_Unwind_Backtrace`.

3.2.3 Network Traffic

A third logging component which is already supported natively by the emulator is capturing of network traffic to a PCAP file. This common format can later be analyzed using tools such as WireShark. In summary, our setup produces three separate log files detailing the app's behavior (see also Figure 1):

- *DroidBox logfile* containing important Java method calls and data from the Dalvik VM;
- *ltrace logfile* containing native method calls in shared objects using JNI;
- *network PCAP file* containing all data sent over the simulated 3G network.

3.2.4 User Interaction

Another issue which has to be considered for dynamic analysis is that of code coverage. For most types of malware, simply launching the app will not trigger any malicious payload - it is necessary for the user to interact with the app and perhaps even confirm some malicious actions.

In order to test a significant fraction of the code paths present in the examined app, we use the *MonkeyRunner* toolkit provided by the Android SDK. Using this utility, it is possible to automatically send simulated interaction events, such as touchscreen contacts or key presses, to the tested app. Since MonkeyRunner does not take any UI elements into account, but rather produces random events, a sufficient number of events should be generated to make sure that most interaction elements have been triggered at least once.

In addition to MonkeyRunner, we also use functionality built into the emulator to simulate external events such as incoming phone calls or SMS messages. These events are sometimes also used by malware to trigger malicious behavior.

3.2.5 Summary

In summary, the following steps are executed in order to analyze the runtime behavior of an app:

1. Reset emulator to the initial state.
2. Launch emulator and wait until startup is completed.
3. Install app to be analyzed using *adb*.
4. Launch app in a new Dalvik VM.
5. Attach *ltrace* to the VM process running the app.
6. Launch *MonkeyRunner* to generate simulated UI events.
7. Simulate additional user events like phone calls.
8. Launch a second run of *MonkeyRunner*.
9. Collect the Dalvik and ltrace log and the PCAP file.

The resulting log files of this process are inserted in our database. This database can be used to perform multiple other analyses.

3.3 Examples

To demonstrate the full range of functions of Mobile-Sandbox and the format of the log files, we now show some results of the log files that resulted from some applications we analyzed. We start with some information from our dynamic analysis which can be very useful when looking at encrypted data inside the application. In this case, the application has encrypted IMEI and IMSI numbers with the help of the DES algorithm before sending these data to a remote server. If you take a look at the network traffic, you would only see a package with encrypted data, but looking in the results from Mobile-Sandbox you get the decrypted data and the DES key that was used for encryption (see Listing 1).

```
Data -- 357242043237517|310005123456789
Algorithm used -- DES
Key -- 77,19,68,124,24,90,10,19,65,16,71,23
```

Listing 1: Decrypted IMEI and IMSI and used encryption key.

While executing the applications inside the emulator we monitored several outgoing SMS messages (see Listing 2 for two examples).

```
Number: 84242 -- Message: QUIZ
Number: 7132 -- Message: 844858
```

Listing 2: Two examples for outgoing SMS messages.

In addition we found several kinds of data leakage. In Listing 3 we displayed the content of a file that was generated by the application.

```
File: /mnt/sdcard/Tencent/v1.log
Operation: write
Data: DeviceInfo [imei=357242043237517,
        telNum=, phModel=generic,
        sysSdk=10, RELEASE=2.3.4]
```

Listing 3: Data leakage to a file located on the unprotected SDcard.

Within our network traffic analysis we found a lot of privacy related data like IMSI, IMEI or smartphone model, which was uploaded to remote servers or web services. One example for such an information leakage via HTTP POST can be seen in Listing 4.

```
<?xml version="1.0" encoding="utf-8"?>
<request>
    <version>1.15</version>
    <platform>2</platform>
    <platformVersion>2.3</platformVersion>
    <IMEI>357242043237517</IMEI>
    <simID>89014103211118510720</simID>
</request>
```

Listing 4: Information leakage found inside recorded network traffic.

4. EVALUATION

We now present an evaluation of our sandbox system. We analyzed the following aspects: correctness, performance, detectability and scalability. At the end of this section we also present a short case study of a malicious app using native calls.

4.1 Correctness

By correctness we mean that an entry in the mobile-sandbox log file only appears if and only if the corresponding action was performed by the analyzed app. To check correctness, we chose 20 samples from a set of malicious applications that we collected from different sources. These samples represent different families of Android malware as shown in Table 1, meant to assure a wide coverage of malicious actions and different points in the development states of malware evolution. More specifically, we chose samples that hit the markets from mid 2010 until beginning of 2012. The LeNa and RootSmart families use exploits and native calls, FakeInst and Adsms send premium SMS messages. The Moghava family acts only on the smartphone itself and modifies locally stored pictures, i.e., there is no malicious action observable that "leaves" the smartphone. The TapSnake

Malware Family	Number of Samples	Primary Usage
Adsms	2	S
BaseBrid	5	I, B
SerBG	2	R, I, B
RootSmart	1	R, I, B, A
LeNa	1	R, I, B, A
Moghava	1	C
FakeInst	7	S
TapSnake	1	L

Table 1: Overview of Mobile Malware used for Evaluation (R = gains root acces, C = compromises local storage, S = sends SMS messages, I = steals privacy related information, B = botnet characteristics, L = steals location data, A = installs additional apps).

family sends location information from the smartphone to a remote server.

Within this sample set, we consider RootSmart to be the most sophisticated malware sample which is, amongst other capabilities, also able to exploit the Android OS while Tapsnake is the simplest sample when looking at the techniques used for malicious behavior.

We manually inspected samples from all these families and consulted all available analysis reports by anti-virus companies and from other sources on the Internet. The resulting action sequences yielded the ground truth to which we compared the behavior the was output by Mobile-Sandbox. Overall, Mobile-Sandbox only detected actions that were part of the ground truth. However, after initial analysis we failed to see certain behaviors that were described in the analysis reports on the Web. Later we realized that the missing behaviors were due to missing external stimuli, i.e., remote servers of a botnet not being active anymore. These insights gave us additional confidence that Mobile-Sandbox is working correctly.

4.2 Performance

The performance of some parts of our system is still rather weak. During the evaluation for this paper we measured runtimes between 9 and 14 minutes for the analysis of one single application. The system is running on an Ubuntu server with Intel Xeon 2,4GHz CPU and 48 GB of RAM.

In average Mobile-Sandbox finishes the virus check within 3 seconds and the subsequent static analysis within additional 8 to 15 seconds. Afterwards the system needs about 2 minutes to reset and reboot a clean version of the emulator. After successfully booting the emulator it takes another 2 to 6 minutes to install the application. This step depends on the file size of the application. The execution of the application and the MonkeyRunner scripts lasts another 6-10 minutes depending on the amount of user events and timers we want to trigger. After shutting down the emulator the system needs additional 10 seconds for analysis of all log files and network traffic.

The performance can be enhanced tremendously by running multiple instances of the analysis frameworks simultaneously.

4.3 Detectability

As we know from malware targeting the Windows environment, there are mechanisms to detect virtualized or sandboxed environments to make the analysis process of the malicious application more difficult or to act differently in these environments. Even if this behavior is not prevalent with Android at the moment, we think that this will change in the future. So, an important security aspect is the detectability of our analysis platform.

Build Information	Emulator	Galaxy S2
Build.BOARD	unknown	smdk4210
Build.DEVICE	generic	GT–I9100
Build.MODEL	sdk	GT–I9100
Build.PRODUCT	sdk	GT–I9100
Build.TAGS	test–keys	release–keys
ro.kernel.qemu	1	0
ro.hardware	goldfish	smdk4210

Table 2: Differences in build information between the emulator and a Samsung Galaxy S2.

A mechanism to detect the Android emulator deals with the specific builds of the operating systems that are used for it. An application querying this information can easily detect if it is running inside an emulator or on a real device. Table 2 shows some system values that can be used for identification as they are sufficiently different from real smartphones. To prevent this detection mechanism a custom build of the Android system is required. In this build we changed the first five variables from Table 2 to the values of a real Samsung Galaxy S2. Unfortunately, modifying the last two values can cause system crashes while running the emulator because there are some Android system services that rely on the fact that these values are set correctly. Another problem hiding the emulator is the fact, that Android launches the `qemud` and `qemu-props` daemons that offer emulation assistance to Qemu when running inside the emulator. Removing these two daemons is not feasible as they are needed to emulate the radio equipment.

Another weak point is Qemu itself. As emulated hardware always behaves differently to native hardware an application is able to detect if it is running inside an emulator by observing the behavior of certain performance aspects of the CPU. Raffetseder et al. [19] show multiple ways to detect the x86 version of Qemu. Similar techniques could also be applied to the ARM architecture to detect the corresponding Qemu version.

Additionally, we changed the default IMSI and IMEI of the emulator (originally both "0") to random values that are consistent with regular IMEI and IMSI numbers. We did this modification to avoid emulator detection mechanisms that check for non-standard or empty values in these device identifiers. We have seen this detection technique employed in various samples, but we are not aware of any other detection mechanisms used in other samples yet.

4.4 Scalability

Our last evaluation criterion refers to the scalability of Mobile-Sandbox, i.e., the question whether it can be used in large-scale analysis projects with several hundreds or thousands of apps.

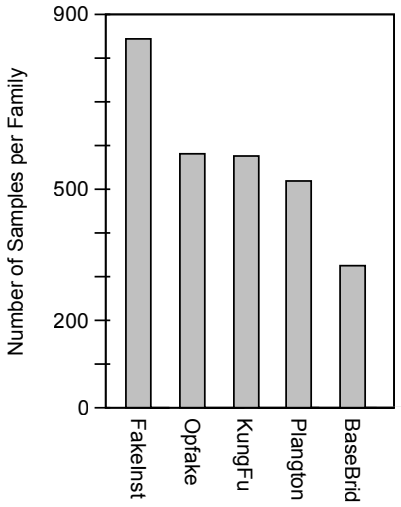
4.4.1 Malware in Third-Party Apps

To evaluate the scalability aspect, we collected about 80,000 apps between December 2011 and August 2012 from the most important Asian markets by downloading them using the Android emulator in an automated fashion. We call this set of apps the "Asian set". We also received about 7,500

Android malware samples from different families through anonymous uploads to our webservice [4] and through the VirusTotal Malware Intelligence Services (VTMIS) [14]. We call this set of samples the "malware set".

We then used Mobile-Sandbox to automatically analyze 36,000 randomly chosen apps from the Asian set and 4,000 randomly chosen samples from the malware set. The analysis results were stored in a database on which we performed statistical analysis. These analyses were performed using a single installation of Mobile-Sandbox within a time span of 14 days.

Figure 2: Top 5 Detected Malware Families and Number of Corresponding Samples in the Union of Asian and Malware Set.



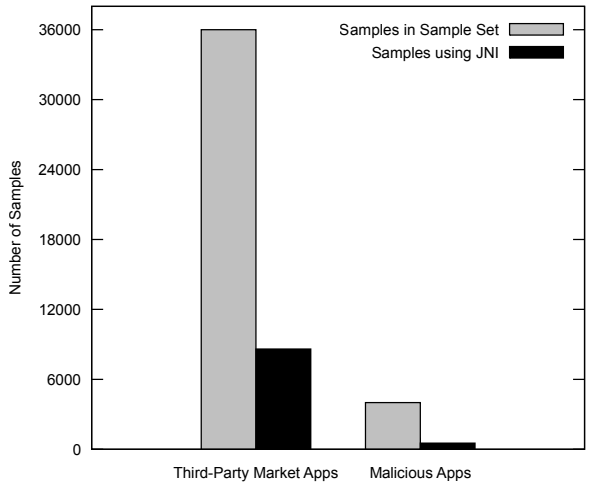
Besides exhibiting that the use of Mobile-Sandbox can scale to several thousand apps, the statistical analysis provides some very interesting results. Considering the union of the Asian set and the malware set, we found 4,641 malicious samples according to the VirusTotal API. Due to the fact, that the malware set consisted of only 4,000 samples, there had to be at least 641 samples from the Asian set that were classified as malicious by VirusTotal. From these 641 samples, 213 samples belong to FakeInst, 185 samples are infected with Opfake, 177 samples are infected with Plankton, 64 samples belong to KungFu and 2 samples belong to the Jifake malware family.

Taking a deeper look at the distribution of the malicious applications we noticed that about 54 percent of the 4,641 samples belong to only four malware families (see Figure 2 for a comprehensive overview). These families are FakeInst, Opfake, KungFu and Plangton. Their main functionality is sending premium SMS messages and, in the case of KungFu, another main functionality is information stealing.

4.4.2 Native Calls

Another point of interest is the use of native interface calls inside Android applications. According to our analysis about 24 percent of the samples from the Asian set use native API calls. When one considers that the author of an app can potentially "hide" many malicious actions inside the native part of the application and that common tools are not able to trace or log this part of the application, the share of one out of four shows why it is so important to develop a system which is able to log these events. Figure 3 also depicts the share of native calling apps in the malware set. Interestingly, this share is about 13 percent. This means that the existence of native calls does not necessarily imply a higher probability that the app is malicious.

Figure 3: Share of Samples Using JNI.



Within the Asian set samples that use native code, we found five samples that were hiding their malicious actions inside the native part of their code. When uploading these samples to VirusTotal we got a detection rate of 0%. This again makes clear how important it is to monitor and analyze native code.

4.4.3 Statistical Implications

In summary, we found 1.78% (641 out of 36,000) samples from the Asian set to be malicious. Recall that 5 out of these 641 were not detected by VirusTotal. But how representative are the results from our measurements? Since we have a rather large measurement base (number of measurements) and we have taken a random sample, we can apply quality assurance techniques from surveys performed in the area of social sciences by Groves [12, 13].

In general, there are two quality criteria for statistical value estimations. The first is the *probability of error*, i.e., the probability that a statement is not true. In empirical research, the probability of error is acceptable if it is below 5%. The second criterion is the *margin of error*, meaning the margin of percentages that the measurement could be different. Acceptable values are 5% or below. Measuring a value of 75% with 5% error probability and 5% error margin, for example, means that with 95% probability, the "real" value is between 70 and 80%.

Following Groves [12, 13], it is sufficient to analyze at least 664 samples to guarantee 1% error probability and 5% error margin. The measurements given above about the use of native code calls in the Asian set can therefore be generalized with high probability. The percentages of malware within the entire Asian set (1.78%) is so small that a 5% error margin is too large to generalize this value. To reach below a 1% error margin we would need to analyze almost 100,000 samples [12, 13] so we cannot generalize our findings in this aspect.

5. CONCLUSIONS

In this paper, we proposed Mobile-Sandbox, a static and dynamic analyzer for Android applications with the purpose to support malware analysts to detect malicious behavior. In the static analysis we parse the application's Manifest file and decompile the application. In a further step we determine if the application is using suspicious looking permissions or intents. The second part of our sandbox performs the dynamic analysis where we execute the application in order to log all performed actions including those stemming from native API calls.

There are still many points to improve Mobile-Sandbox, especially regarding the performance. For a better performance and thus, for analyzing more samples in an appropriate time period, an idea would be to parallelize the analysis process and try to fix the crashes of MonkeyRunner. These improvements will probably also lead to a more reliable system. As an additional future work, we want to implement a malware detection system which is no longer based on antivirus systems: The idea is to use our results from Section 4 and combine them with machine learning techniques based on our 1TB sample set. With the help of these techniques, we hope to find new heuristics and patterns for efficient malware detection and clustering.

For all the mobile users we will provide an application for the Android platform which is able to check the status of already installed applications and is able to upload them directly into our sandbox for analysis if no report is available.

Acknowledgments: This work has been supported by the Federal Ministry of Education and Research (grant 01BY1021 – MobWorm).

6. REFERENCES

[1] Android Developers. Using the Android Emulator, January 2012.

[2] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. Camtepe, and S. Albayrak. An android application sandbox system for suspicious software detection. *In Proc. of the 5th International Conference on Malicious and Unwanted Software (MALWARE)*, 2010.

[3] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. *In Proc. of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, 2011.

[4] Department of Computer Science Friedrich-Alexander-University Erlangen-Nuremberg. Mobile-Sandbox. `http://www.mobile-sandbox.com`, January 2012.

[5] A. Desnos. Androguard, January 2011.

[6] A. Desnos and G. Gueguen. Android: From reversing to decompilation. *In Proc. of Black Hat Abu Dhabi*, 2011.

[7] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *In Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2010.

[8] A. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14. ACM, 2011.

[9] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. *In Proc. of the 18th ACM conference on Computer and communications security*, 2011.

[10] J. Freke. smali - an disassembler for android's dex format, September 2009.

[11] Google Inc. Android SDK, October 2009.

[12] R. M. Groves. Research on survey data quality. *Public Opinion Quarterly*, 51(2):157–172, 1987.

[13] R. M. Groves. *Survey Errors and Survey Costs*. Wiley, 1989.

[14] Hispasec Sistemas S.L. VirusTotal Malware Intelligence Services.

[15] Hispasec Sistemas S.L. VirusTotal Public API.

[16] P. Lantz. droidbox - android application sandbox, February 2011.

[17] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. *In Proc. of the 23rd Annual Computer Security Applications Conference*, 2007.

[18] V. U. of Technology. Andrubis - analysis of android apks. `http://anubis.iseclab.org`, May 2012.

[19] T. Raffetseder, C. Kruegel, and E. Kirda. Detecting system emulators. In *ISC*, pages 1–18, 2007.

[20] A.-D. Schmidt, R. Bye, H.-G. Schmidt, J. Clausen, O. Kiraz, K. Yüksel, S. Camtepe, and A. Sahin. Static analysis of executables for collaborative malware detection on android. *In Proc. of the ICC Communication and Information Systems Security Symposium*, 2009.

[21] J. Six. *Application Security for the Android Platform: Processes, Permissions, and Other Safeguards*. Oreilly & Assoc Inc, 2011.

[22] M. Spreitzenbarth. The Evil Inside a Droid - Android Malware: Past, Present and Future. In *Proceedings of the BALTIC CONFERENCE Network Security and Forensics*, 2012.

[23] The Debian Project. ltrace, January 2012.

[24] C. Willems and F. C. Freiling. Reverse code engineering - state of the art and countermeasures. *it - Information Technology*, pages 53–63, 2011.

[25] C. Willems, T. Holz, and F. C. Freiling. Toward automated dynamic malware analysis using CWSandbox. *IEEE Security & Privacy*, 5(2):32–39, 2007.

[26] L. Xie, X. Zhang, J.-P. Seifert, and S. Zhu. pbmds: a behavior-based malware detection system for cellphone devices. *In Proc. of the third ACM conference on Wireless network security*, 2010.

[27] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. *In Proc. of the 33rd IEEE Symposium on Security and Privacy (Oakland 2012)*, May 2012.

[28] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. *In Proc. of the 19th Annual Symposium on Network and Distributed System Security*, 2012.