An Android Application Sandbox System for Suspicious Software Detection

Thomas Bläsing, Leonid Batyuk, Aubrey-Derrick Schmidt,
Seyit Ahmet Camtepe, and Sahin Albayrak
Technische Universität Berlin - DAI-Labor
{thomas.blaesing, leonid.batyuk, aubrey.schmidt}@dai-labor.de
{ahmet.camtepe, sahin.albayrak}@dai-labor.de

Abstract

Smartphones are steadily gaining popularity, creating new application areas as their capabilities increase in terms of computational power, sensors and communication. Emerging new features of mobile devices give opportunity to new threats. Android is one of the newer operating systems targeting smartphones. While being based on a Linux kernel, Android has unique properties and specific limitations due to its mobile nature. This makes it harder to detect and react upon malware attacks if using conventional techniques.

In this paper, we propose an Android Application Sandbox (AASandbox) which is able to perform both static and dynamic analysis on Android programs to automatically detect suspicious applications. Static analysis scans the software for malicious patterns without installing it. Dynamic analysis executes the application in a fully isolated environment, i.e. sandbox*, which intervenes and logs low-level interactions with the system for further analysis. Both the sandbox and the detection algorithms can be deployed* in the cloud*, providing a fast and distributed detection of suspicious software in a mobile software store akin to Google's Android Market. Additionally, AASandbox might be used to improve the efficiency of classical anti-virus applications available for the Android operating system.*

1 Introduction

Anti-virus research is an ongoing process of identifying and analysing new and unknown malware for extracting possible detection schemes that can be used within anti-virus software. A virus scanner [33] can block viruses, worms, and Trojan horses from infecting the often real time monitored system. Although behavioural detection is mostly included in anti-virus software, most malware is detected by scanning for and finding a certain patterns, also called signatures. Therefore, the malware has to be known by the scanner through a signature database, otherwise it is not able to detect and remove it. For generating these signatures, malware needs to be analysed for identifying significant and meaningful patterns that should be unique to the malware and its relatives[1]. But not only signature-generation requires analysis; in order to disinfect a system, the behaviour of the malware needs to be understood for being able to undo critical changes. For doing so, sandboxes can be used, which guarantees an almost realistic execution of suspicious binaries in an isolated environment. Such sandboxes are known in the domain of stationary computers, e.g. CWSandbox [34] or Java sandboxes [12], but especially smartphone platforms lack such kind of analysing software[2]. This raises concerns since smartphones have already faced a wave of malware [31, 28] and it can be expected that new malware will emerge for popular platforms, like iPhone[3] or Android[4]. In this case, analyzing capabilities are left to a few people within anti-virus companies. This would not be that concerning but Oberheide et al. [23] showed that the average time for receiving a signature for a new malware is about 48 days. This in turn means that users with infected system need to wait 48 days until they have a chance to disinfect it, leaving the window of opportunity very wide open for new malware.

Therefore, we present an approach how to realize a sandbox for Android-based applications. We preferred Android over iPhone since its source code is set open source allowing us to implement modifications even on system level of the operating system. Our sandbox is capable of performing static and dynamic analysis. In the static part, the sandbox decompresses installation files and disassembles corresponding executables. This can be used for *cheap* and fast pre-checks that might already indicate malicious code fragments and characteristics. In the dynamic part, we make use

[1] malware families contain malware that share similar malicious code fragments

[2] anti-virus vendor internal tools are not published

[3] http://www.apple.com/iphone

[4] http://www.android.com

978-1-4244-9356-2/10/$26.00

of the Android emulator which is normally used for testing and debugging ordinary Android applications. Investigated Applications are installed to the emulated and isolated environment. After that, applications are executed and can be used within the sandbox for performing behavioural analysis. For improving the dynamic analysis process, the possibility of automated generation of user inputs is investigated. Since these analyses requires extensive resource capabilities, our system is intended to be run as a cloud service. Software distributors, like the Android Market or the AppStore, can run this analyses on each submitted application or users, in turn, can upload suspicious applications to their convenience.

This work is structured as follows. Section 2 gives more insights into the field of Smartphone Security. In Section 2.2, we describe common analysis methodologies used to investigate malicious software. In Section 3, our approach in creating such a sandbox is presented. Section 4 presents possible analysing procedures that can be performed in our sandbox. In Section 6, we conclude.

2 Related Work

In the following section, we give a short introduction to the field of mobile malware research, and an overview on static and dynamic malware detection techniques, including sandboxing.

2.1 Smartphone Security

In developed countries, mobile phones often outnumber potential subscribers. For example, in Germany over 80 million handsets are spread over a total of approximately 82 million residents [27]. According to [11], the figures in other countries are similar, and the share of smartphones is increasing. While many phones are being primarily used for making calls and sending text messages, browsing the internet and using third-party applications gained a significant share in mobile device usage. This trend clearly requires a deeper look at the security aspects of mobile platforms, especially those which allow internet usage along with installing third-party applications. These are commonly referred to as *smartphones*.

In order to provide most versatility and flexibility to the user, smartphones run sophisticated operating systems comparable to those of desktop PCs. Manufacturers provide developers with tools to create and distribute custom applications. Due to high complexity of mobile device software, security threats emerge [14], comparable to the those which a conventional PC is exposed to. Nevertheless, smartphone security has its own specifics [31]. A smartphone is tied to a mobile network infrastructure, including its billing system, providing an attacker with a means of immediate commercial exploitation, making identity theft possible, and allowing for Denial-of-Service attacks against the (potentially fragile) wireless network [4]. Additionally, privacy issues arise since a smartphone can deliver a lot of delicate information about its user straight out of its sensors, such as location, microphone, camera, etc.

Since smartphones are a viable target to attackers, sufficient defence mechanisms must be introduced. However, this is a hard task, partly due to limitations of mobile hardware which is not capable of running a full blown malware detection suite in the background. Another reason is, according to [31], that this research field gained insufficient attention, especially when compared to conventional desktop- and server-based malware detection.

Only a few good works cover security issues of mobile platforms. Most authors focus on the attack mechanisms. Collin Mulliner et. al. [19, 18, 20, 21] developed several smartphone exploits, both on Windows Mobile and SymbianOS. Racic et al. [24] discovered an MMS vulnerability which allows an attacker to drain the battery of the victim's phone. Becher et. al. [1] created a worm running on a version of Windows Mobile.

However, despite the fact that Android is a young mobile OS if compared to Apple's iOS or Nokia's Symbian, several publications discuss Android-specific security mechanisms, involving overall security assessment of the platform [5, 31], malware detection [29], application permission analysis [8], and kernel hardening [32]. This amount of attention to one particular mobile platform clearly indicates the emerging need in solid mechanisms for security analysis on Android.

2.2 Static Software Analysis vs. Sandboxing Techniques

For malware detection, a detailed knowledge of application's characteristics is essential, which may be obtained by various means. According to [2], two common practices exist - *static* and *dynamic* analysis of software. Both have advantages and disadvantages [17], and numerous approaches to both static and dynamic analysis paradigms exist.

Static analysis involves various binary forensic techniques, including decompilation, decryption, pattern matching [33] and static system call analysis [30]. All of these techniques have in common that the (potentially malicious) software is not being executed. Here, a common approach is filtering binaries by malicious patterns, called *signatures*. Even nowadays, many anti-virus software suites base on this approach. Due to its nature, static analysis has an advantage of being fast and relatively simple [17]. Its primary disadvantage is that malicious code patterns have to be known *in advance*, making it impossible to automatically detect new malware or malicious polymorphic code without

an intervention of a human expert.

A set of techniques, which involve running an application in a controlled environment and monitoring its behaviour, are known as *dynamic analysis*. Various heuristics have been used to best capture and analyse dynamic behaviour of applications, such as monitoring file changes, network activity, processes and threads [33] and system call tracing [9], on which we focus in our work. Since software is being executed and kept alive for a longer time to obtain its runtime characteristics, it is obvious that dynamic analysis techniques are computationally more complex than static analysis.

A common approach to dynamic software analysis is *sandboxing*. A *sandbox* can be defined as "an environment in which the actions of a process are restricted according to a security policy" [3]. In practice, this means that a sandbox is an instance of the target OS, which is isolated in a way that prevents malware from performing harmful actions.

A sandbox' security policy may vary, depending on the actual use case. The policy may be defined in a defensive way, causing the process to be stopped at a particular point in order to be examined - or even aborted to prevent potential damage to the system. Another option is monitoring and recording all system activity while the application is running, and processing the gathered information afterwards. This approach is especially useful for monitoring of unknown software and classifying it, e.g. by identifying abnormal system states and analysing their cause [22, 13].

Several designs and implementations of application sandboxes exist, such as the widely used CWSandbox [34]. Existing sandboxes differ in the targeted platform, and the way policy enforcement is performed. *User space* sandbox systems rather inject code into system libraries in order to gain control over the APIs which are used by most applications, or run software samples through a debugger, e.g. ADSandbox [6]. User space sandboxes can be detected by malware, which, in turn, would simulate normal behaviour in order to remain undetected [25]. *Kernel space* sandboxes try to overcome this problem by taking a low-level approach of monitoring raw system calls *inside the kernel*. This minimizes the chance of the sandbox being recognized by malware, but is harder to implement and produces large amounts of highly complex data. However, even kernel-level root-kit-based sandboxes can possibly be detected, as shown in [15, 7, 10].

Unfortunately, the majority of existing sandbox implementations target desktop and server operating systems, due to their heterogeneity and wide deployment. There has been very little to no research on sandboxes targeting mobile environments such as iOS or Android. Our main contribution tries to fill this gap and provide a robust set of tools for malware analysis on Android, including polymorphic viruses and new threats.

3 Android Application Sandbox

Sandboxes are often located within kernel space since access to critical parts of the OS can be realized. The kernel is a very essential part of a system because it acts as bridge between hardware and software.

One approach of sandbox systems is to monitor system and library calls including their arguments. This is often done trough system call redirecting, also known as system call hijacking [35]. System calls, short system calls, are function invocations made from user space into the kernel in order to request some services or resources from the operating system [16]. For understanding how system calls can be hijacked, we will first explain how they are invoked in general. Figure 2 shows an example for the `read()` system call on a Linux based system. In turn, Figure 1 shows the same example from Figure 2 but here the system call is finally redirected.

For being capable to analyze potential harmful applications for the Android platform our system realizes system and library call monitoring. Android uses a modified Linux basis to host a Java-based middleware running the user applications. This implies that calls should not be monitored on Java level since other calls being made by native Linux application might get lost. One might argue that only Java user application space is relevant for malware detection but Schmidt et al. [31] showed that it is possible to place piggybag Linux malware into Android systems. Due to this nature, calls are monitored on the lowest level possible. For accomplishing this task, we have implemented a loadable kernel module (LKM) being placed in the Android emulator environment. This action needs special attention since the emulator tends to get very unstable from low-level changes, like we realized. Especially resource limitations and modifications of the Linux System by Google need to be treated carefully, which e.g. also had results on the amount of data being traced. Currently, the system is not able to store arguments of calls since this takes too long causing the system to crash. Additionally, Android bases on an ARM architecture leading to the problem that the ID of the calls are different to the ones, e.g. from an Intel system.

The LKM is intended to hijack all available system calls targeting for the ARM architecture since this is used for the Android operating system. The following pseudo code shows how the system calls are redirected:

```
asmlinkage long
new_syscall
(type1 param1, type2 param2, ...)
{
  retval = orig_syscall(type1 param1, type2 param2, ...);
  PRINT_TO_LOG( "syscall()", retval);
  return retval;
}
```

In the pseudo code you can see that each system call which is redirected firstly behaved like the original system

call to get the real return value. This is achieved since because we have a policy that enforces only logging of the reactions of the system call. Then the "PRINT_TO_LOG" prints out the gathered informations to the kernel log. After this, the original return value is returned to the kernel environment so that the context of the application can proceed.

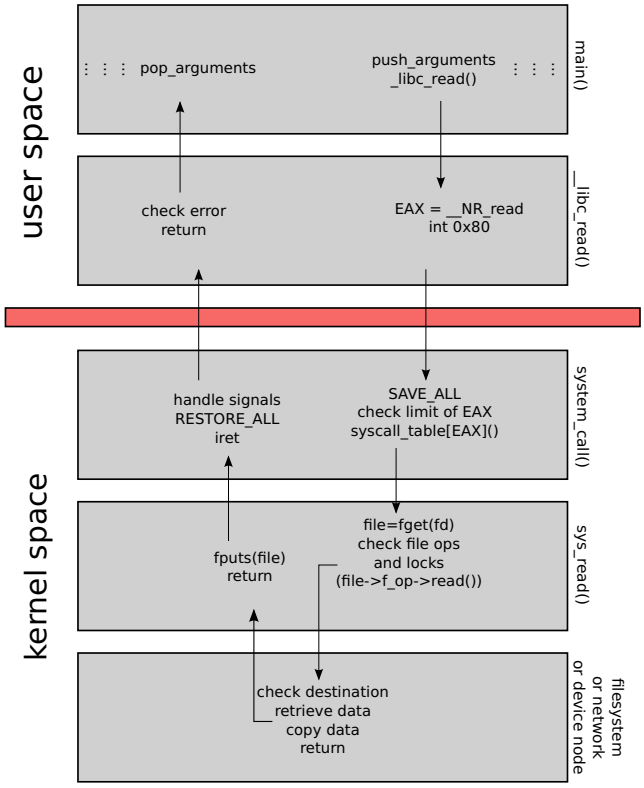

Figure 2. Steps involved in performing a `read()` system call from user space (derived from [26]). Each arrow in the figure represents a jump in the instruction flow.

4 Static and Dynamic Analysis of Android Applications

We propose a novel two-step analysis of Android applications, consisting a full-fledged kernel-space sandbox, and a fast static pre-check. AASandbox executes automatically, without any need for human interaction, and saves the logs of system calls and static analysis for further inspection.

As an input, the AASandbox takes an Android application archive, which is packaged in a `*.apk` file and is therefore referred to as *APK*. Applications are written in Java and run in a custom Java virtual machine called Dalvik. Application source code is first compiled to standard Java bytecode, and then optimized and converted to Dalvik executable format for being interpreted Dalvik VM. Bytecode is then packaged together with other application resources, including UI layouts, localization and a manifest file which defines the structure of the application. The structure of an APK package is depicted in Figure 3.

The AASandbox first performs a static pre-check, followed by a full-blown dynamic analysis.

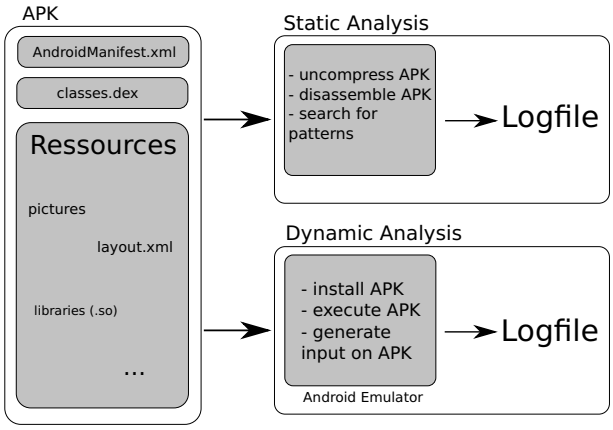

Figure 3. Design of the Android Application Sandbox (AASandbox). AASandbox consists of three main parts: the APK, static and dynamic analysis methods, and resulting dataset for further analysis.

4.1 Static Analysis of Android Applications

An Android application package (APK) is scanned for special patterns (e.g. *Runtime.Exec()*), which may be used to classify the application. Our implementation of static analysis is lightweight enough to run on-device. However, since the dynamic analysis clearly requires emulation on a more powerful machine, we decided to run the static analysis off-line, too.

Decompression: An Android application is a compressed (ZIP) bundle of files. When uncompressed, its content is split into three main parts in a directory named `APK-NAME/unzipped/`:

- `AndroidManifest.xml` - an XML file holding meta information about the application, e.g. descriptions, security permissions, etc.
- `classes.dex` - a single file which holds the complete bytecode to be interpreted by Dalvik VM.
- `res/` - a folder consisting of files defining the layout, language, etc.

Getting the Starter Name: In this step, the main "launchable activity" is extracted from the manifest file. It is used for identifying the application, but is also important

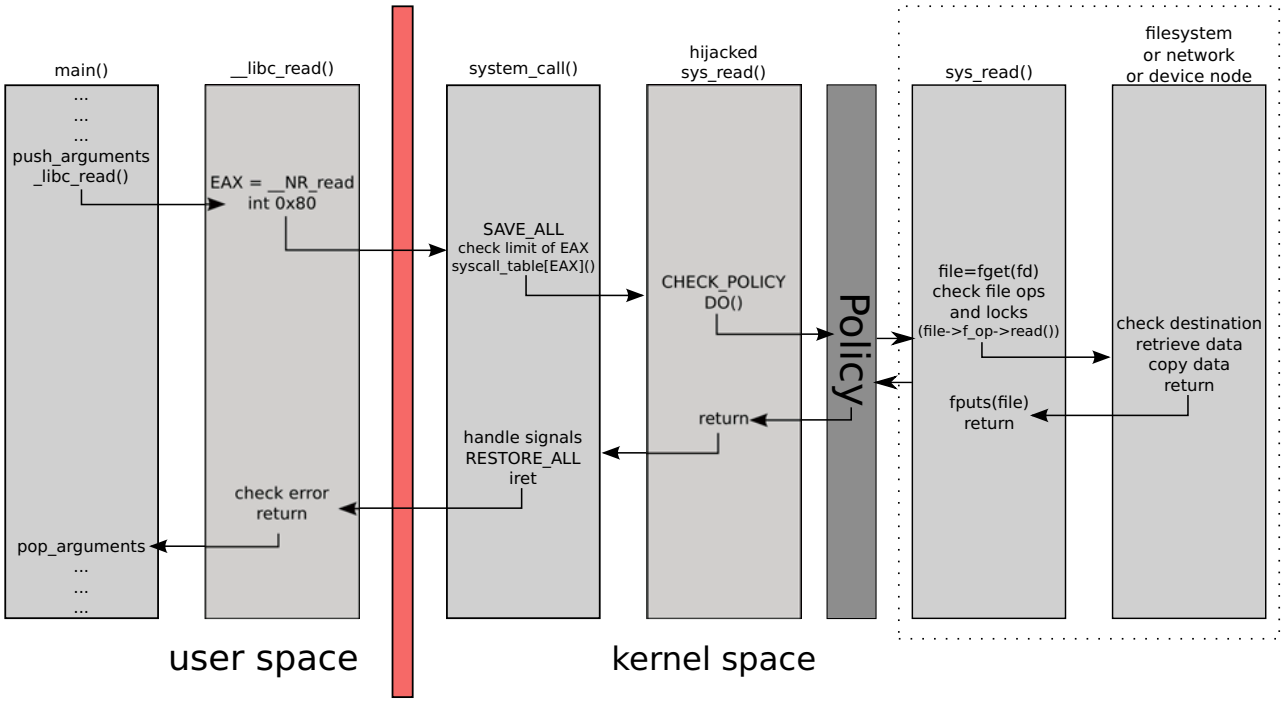

Figure 1. Steps involved in performing a hijacked read() system call. The procedure on the first steps is similar to Figure 2 up to the step when the hijacked read() system call routine arrives. There, the further processing is described by the policy and depends on the implementation of the system call itself.

later for the dynamic analysis, since it needs an entry point to the application UI.

Decompilation: The file classes.dex holds the actual bytecode of the application. It is converted into a human-readable format using Baksmali [5], which produces a Java-typical folder hierarchy containing files with easily-parsable pseudo-code.

Pattern Search: Finally, disassembled code can be be scanned for suspicious patterns. Due to the absence of sandboxes for Android, we needed to find out which pattern might indicate presence of potentially harmful application. Therefore, we started a trial-and-error approach using as much as possible indicators that were available. In a first attempt, we tried to cluster extracted patterns in order to find a relevant set of indicators. In our case *relevant* means that applications are assigned to clusters corresponding to their *activity* which was known due to a labeled set of applications. The results of our approach led to the following patterns we currently scan for:

- usage of the Java Native Interface, which can be used to dynamically load native libraries.
- usage of `Sytem.getRuntime().exec(..)`, which can be used to spawn native children processes and surpass the normal application life-cycle
- usage of reflection (2 patterns total), which may be used to circumvent API restrictions
- usage of services and IPC provision, which may drain the battery or overload the device's CPU
- usage of Android Permissions to determine which permissions will be granted at install time

[5] A disassembler for the DEX format as used by the Dalvik VM, `http://code.google.com/p/smali/`

4.2 Dynamic Analysis of Android Applications

Dynamic analysis is usually more complex than the static analysis. In this work, the application is installed in the standard Android Emulator from the Google Android SDK bundle. After installation it will be executed for a specific time and penetrated with random user inputs from the "Android Monkey" tool. The Monkey is a program that runs on the emulator or device and generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events. The Monkey was primarily invented to stress-test applications. Our Android Application Sandbox (AASandbox) is placed in kernel space and hijacks system calls for logging. AASandbox helps us to log the behaviour of the application at the sys-

tem level. The resulting log file will be then summarized and reduced to a mathematical vector for better analysis.

The kernel module makes sure that each occurrence of a system call is logged, including those which are performed by processes other than the currently analysed application. This makes sure that a complete system state is recorded and no malicious activity can be hidden. The log of system calls for an application is stored in a separated file.

Dynamic analysis includes following steps:

Prepare and Start Emulator: The Android SDK includes a mobile device emulator which runs on a normal desktop computer. It emulates all of the hardware and software features of a typical mobile device, except doing phone calls. For testing and modelling an application, the emulator supports Android Virtual Device (AVD) configurations. AVDs specify the Android platform that the emulator shall run, as well as the hardware options and emulator skin files which shall being used. Once an application is running on the emulator, it can use the services of the Android platform to invoke other applications, access the network, play audio and video, store and retrieve data, notify the user, and render graphical transitions and themes. The emulator also includes a variety of debug capabilities, such as a console from which the kernel output can be logged, simulate application interrupts (such as receiving SMS messages or phone calls), and simulate latency effects and packet drops on the data channel.

Install AASandbox: Goal of the dynamic analysis is to examine the system state changes which happens when a given application is executed. To reach this the AASandbox have a log-only policy which does not actively intercept any system state change. Therefore, we developed a loadable kernel module (LKM) which assures the policy given by the sandbox environment. The insertion of the LKM into the running kernel of the Android device emulator is done with the help of Android Debugging Bridge (ADB) which comes along with the SDK from Android. Once the LKM is loaded the produced output is sent to a logfile.

Install APK and start Monkey: The installation of the given APK is also done with the help of ADB. Here the ADB copies the APK into the image of the emulator and then runs the PackageManager which is an essential part of Android and installs it onto the system. This means that the APK will be unzipped and copied to the specified directories. After installation the application is imported into the Android system and can be manually started through the main applications menu. To automate this for AASandbox the application will be launched with the help of the Android Monkey. On AASandbox the Monkey has the role of a simulator of human interaction on the to examined application. During the runtime of the Monkey, there are exactly 500 generated events with 1000 ms silence in between.

Obtain system call logs: When the Monkey is finished the mobile device emulator process is killed and the used AVD configuration which was created on step 1 is removed.

5 Experiments

To proof the correct working of the whole system we will now describe an example run of the AASandbox. The example application we are using here is a self-written fork bomb which uses Runtime.Exec() to start an external binary program. The application creates subprocesses of itself in an infinite loop. The intended behaviour is that the operating system is not responding after a while. This kind of attack is typically known as Denial of Service (DoS) and is an typical example of an attack.

Starting the whole process is as simple as starting the shell script `aas.sh` with the desired application package name as parameter. The script will then start the first and second analysis step of the AASandbox automatically.

During the first static analysis step an directory `REPORTS/ForkBomb.apk/` is created. There are also two subdirectories, `unzipped/` and `disasm/`, created which contains the unzipped application package and the disassembled bytecode:

```
REPORTS/ForkBomb.apk/
|-- disasm
|   |-- AndroidManifest.readable
|   `-- de
|       `-- dailab
|           `-- ccsec
|               `-- forkbomb
|                   |-- ForkBombActivity.smali
|                   |-- R$attr.smali
|                   |-- R$drawable.smali
|                   |-- R$layout.smali
|                   |-- R$raw.smali
|                   |-- R$string.smali
|                   `-- R.smali
`-- unzipped
    |-- AndroidManifest.xml
    |-- META-INF
    |   |-- CERT.RSA
    |   |-- CERT.SF
    |   `-- MANIFEST.MF
    |-- classes.dex
    |-- res
    |   |-- drawable
    |   |   `-- icon.png
    |   |-- layout
    |   |   `-- main.xml
    |   `-- raw
    |       `-- forkbomb
    `-- resources.arsc
```

After unpacking and disassembling is done the desired files are scanned for static patterns like an signature-based anti-virus application will do. One of the main features we will scan for are security related, like use of reflection or executing external programs, and which permissions will be used by the application itself. The permission scanning is very important because without a granted permission on Android it is not possible to do the intended action.

The logfile output for the first step is in the following format:

```
$ cat LOGS/ForkBomb.apk-s1.log
000100000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
00000000
```

The output shows that the fork bomb application is detected for using `Runtime.Exec()` to start an external program. The next interesting point is that the program do not use any possible Android permission.

After this first step is completed the dynamic analysis of the application starts. There, the prepared Android emulator will start as described in Section 4.2. Once the emulator is up and running the application will be installed via `adb install ForkBomb.apk`. After the installation is completed the Android Monkey will be started via `adb shell monkey -p $ACTIVITY -vv --throttle 1000 500`. This tells the Monkey to start the activity associated with the fork bomb application and generate 500 random user events which will be used to simulate normal user behaviour.

All the produced output of the emulator run will be logged into `LOGS/ForkBomb.apk-s2.log` in the following format:

```
found sct at 0xc0022f04
RK loaded
[1272980618.635790][recvfrom()-83][1;0]
[1272980618.636043][recvfrom()--11][1;0]
[1272980618.638795][munmap()-0][193;0]
[1272980618.640029][sigprocmask()-0][192;0]
[1272980618.641172][wait4()-192][39;0]
[1272980618.641429][read()--5][39;0]
[1272980618.641622][write()-4][39;0]
[1272980618.642570][close()-0][39;0]
[1272980618.643198][write()-24][81;0]
[1272980618.645523][write()-4][82;0]
[1272980618.712968][read()-61][37;0]
...
```

The first value is a timestamp, the next is the used system call followed by the original return value. The last two numbers are the IDs of the process and its parent. Along with this, information it is now possible to create a system call histogram which looks like this:

```
1499, 0, 4, 0, 0, 3, 79, 6, 0, 37, 0, 0, 0, 0, 0, 0, 0,
4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 3, 0, 1, 0, 1, 0,
0, 28, 0, 0, 0, 0, 0, 0, 0, 0, 1488, 0, 0, 0, 0, 0, 0, 0,
0, 0, 1, 0, 0, 0, 12, 29, 3, 0, 0, 0, 1, 0, 13, 0, 0, 596,
0, 0, 0, 0, 2, 1, 0, 1, 0, 0, 2, 0, 0, 0, 4, 0, 0, 0, 373,
1, 1, 0, 0, 1, 34, 0, 372, 34, 1, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 225, 1420, 0, 0, 2, 90, 0, 0, 0, 0, 50,
0, 0, 0, 0, 13, 0, 8, 6, 7, 2, 112, 112, 13, 9, 0, 452, 48,
0, 0, 0, 0, 0, 0, 0, 3, 4, 438, 359, 50, 0, 0, 0, 0, 3, 3,
1157, 6411, 0, 0, 670, 47, 0, 0, 2, 2, 0, 0, 0, 0, 0, 6, 6,
0, 0, 2, 3, 3, 0, 0, 13, 3, 10, 21, 9, 12, 5, 1, 0, 1, 334,
0, 0, 0, 134, 200, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 19, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3
```

Here the counting of all syscalls are shown. For example the first syscall (`time`) is used 1499 times where the second syscall (`stime`) is not used at any time throughout application runtime.

We used about 150 applications in order to test and evaluate our system. These 150 programs were collected through the official Android Market representing the top 150 popular applications in October 2009. These applications were used to test the system showing positive results in the clustering. For testing malware, we used our self-written fork bomb. As soon as more malware is available to us, more tests will be performed. For the current status of the system, malware characteristics and behavior known from other smartphone platforms, e.g. Symbian OS, are considered to be analyzed in our sand box. Comprehensive malware techniques, like polymorphy and code fragment encryption, are not regarded due the early stage of the system.

The fork bomb malware clearly appears to be an outlier among the investigated applications. In general, our first impression of the data obtained by the AASandbox is that the measurements are very diverse, delivering a very high entropy dataset. We believe that, after further investigation, the data can be sufficient for detecting new malware. Our aim is to achieve this in future by employing various machine-learning techniques.

6 Conclusion

In this work we presented a sandbox created for analysing Android applications applicable as cloud service. Therefore, we showed how the Android emulator can be used to run Android applications in an isolated environment. Unlike other sandboxes, we added a pre-check functionality that can analyse Android executables in a static manner. This can indicate usage of malicious patterns within source code. In the dynamic analysis, system calls can be traced and corresponding reports are logged. These can be used for further investigations, either performed manually or automatically.

References

[1] M. Becher, F. Freiling, and B. Leider. On the effort to create smartphone worms in windows mobile. In *Information Assurance and Security Workshop, 2007. IAW '07. IEEESMC*, pages 199–206, 20-22 June 2007.

[2] J. Bergeron, M. Debbabi, J. Desharnais, M. M. Erhioui, Y. Lavoie, and N. Tawbi. Static detection of malicious code in executable programs. In *Proceedings of the Symposium on Requirements Engineering for Information Security (SREIS'01)*, 2001.

[3] M. A. Bishop. *The Art and Science of Computer Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[4] Bundesamt für Sicherheit in der Informationstechnik. Mobile endgeräte und mobile applikationen: Sicherheitsgefährdungen und schutzmassnahmen, 2006.

[5] J. Burns. Developing secure mobile applications for android - an introduction to making secure android applications. https://www.isecpartners.com/files/iSEC_Securing_Android_Apps.pdf, 2008. [Online; accessed 01-March-2010].

[6] A. Dewald, T. Holz, and F. Freiling. Adsandbox: Sandboxing javascript to fight malicious websites. In *Symposium on Applied Computing (SAC) 2010, Sierre, Switzerland*, march 2010.

[7] M. Dornseif, T. Holz, and C. Klein. Nosebreak - attacking honeynets. In *In Proceedings of the 2004 IEEE Information Assurance Workshop*, 2004.

[8] W. Enck, M. Ongtang, and P. McDaniel. Understanding android security. *IEEE Security and Privacy*, 7(1):50–57, 2009.

[9] S. Forrest, S. Hofmeyr, and A. Somayaji. The evolution of system-call monitoring. In *ACSAC '08: Proceedings of the 2008 Annual Computer Security Applications Conference*, pages 418–430. IEEE Computer Society, 2008.

[10] T. Garfinkel. Traps and pitfalls: Practical problems in in system call interposition based security tools. In *Proc. Network and Distributed Systems Security Symposium*, 2003.

[11] D. K. Goldhammer, D. A. Wiegand, D. Becker, and M. Schmid. Goldmedia mobile life report 2012, mobile life in the 21st century, status quo and outlook. http://www.bitkom.org/60376.aspx?url=081009_bitkom_goldmedia_mobile_life_2012(1).pdf. [Online; accessed 01-May-2010].

[12] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit™ 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 102–112, Monterey, California, Dec. 1997.

[13] D. Gopan and T. Reps. Low-level library analysis and summarization. In *In CAV*, pages 68–81, 2007.

[14] G. Lawton. Is it finally time to worry about mobile malware? *Computer*, 41(5):12–14, 2008.

[15] J. Levine, J. Grizzard, and H. Owen. A methodology to detect and characterize kernel level rootkit exploits involving redirection of the system call table. In *IWIA '04: Proceedings of the Second IEEE International Information Assurance Workshop (IWIA'04)*, page 107. IEEE Computer Society, 2004.

[16] R. Love. *Linux System Programming*. O'Reilly, 2007.

[17] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Proceedings of the 23rd Annual Computer Security Application Conference (ACSAC)*, pages 421–430, 2007.

[18] C. Mulliner. Exploiting pocketpc, 2005. Talk on WhatTheHack 2005, http://wiki.whatthehack.org/images/c/c0/Collinmulliner_wth2005_exploiting_pocketpc.pdf.

[19] C. Mulliner. Advanced attacks against pocketpc phones. 2006.

[20] C. Mulliner. Exploiting symbian: Symbian exploitation and shellcode development. http://mulliner.org/symbian/feed/CollinMulliner_Exploiting_Symbian_BlackHat_Japan_2008.pdf, 2008. Talk on BlackHat Japan 2008, visited 15.6.2009.

[21] C. Mulliner and G. Vigna. Vulnerability analysis of mms user agents. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, pages 77–88, Washington, DC, USA, 2006. IEEE Computer Society.

[22] D. Mutz, W. K. Robertson, G. Vigna, and R. A. Kemmerer. Exploiting execution context for the detection of anomalous system calls. In *RAID*, pages 1–20, 2007.

[23] J. Oberheide, E. Cooke, and F. Jahanian. Cloudav: N-version antivirus in the network cloud. In *Proceedings of the 17th USENIX Security Symposium (Security'08)*, San Jose, CA, July 2008.

[24] R. Racic, D. Ma, and H. Chen. Exploiting mms vulnerabilities to stealthily exhaust mobile phone's battery. In *Proceedings of the Second IEEE Communications Society / CreateNet International Conference on Security and Privacy in Communication Networks (SecureComm), Baltimore, MD*, Aug. 2006.

[25] T. Raffetseder, C. Kruegel, and E. Kirda. Detecting system emulators.

[26] A. Rubini. Kernel system calls. http://www.ar.linux.it/docs/ksys/ksys.html. [Online; accessed 01-March-2010].

[27] A. Schmidt and S. Albayrak. Malicious software for smartphones. Technical Report TUB-DAI 02/08-01, Technische Universität Berlin, DAI-Labor, Feb. 2008. http://www.dai-labor.de.

[28] A.-D. Schmidt and S. Albayrak. Malicious software for smartphones. Technical Report TUB-DAI 02/08-01, Technische Universität Berlin - DAI-Labor, Feb. 2008.

[29] A.-D. Schmidt, R. Bye, H.-G. Schmidt, J. Clausen, O. Kiraz, K. Yüksel, A. Camtepe, and S. Albayrak. Static analysis of executables for collaborative malware detection on android. In *IEEE International Congress on Communication (ICC) 2009 - Communication and Information Systems Security Symposium*, 2009.

[30] A.-D. Schmidt, J. H. Clausen, S. A. Camtepe, and S. Albayrak. Detecting symbian os malware through static function call analysis. In *Proceedings of the 4th IEEE International Conference on Malicious and Unwanted Software (Malware 2009)*, pages 15–22. IEEE, 2009.

[31] A.-D. Schmidt, H.-G. Schmidt, L. Batyuk, J. H. Clausen, S. A. Camtepe, S. Albayrak, and C. Yildizli. Smartphone malware evolution revisited: Android next target? In *Proceedings of the 4th IEEE International Conference on Malicious and Unwanted Software (Malware 2009)*, pages 1–7. IEEE, 2009.

[32] A. Shabtai, Y. Fledel, and Y. Elovici. Securing android-powered mobile devices using selinux. *IEEE Security and Privacy*, 99(PrePrints), 2009.

[33] P. Szor. *Virus Research and Defense*. Addison Wesley, 2005.

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

[35] D. D. Zovi. Kernel rootkits.