Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

A game of Droid and Mouse: The threat of split-personality malware on Android

Dominik Maier, Mykola Protsenko, Tilo Müller *

Department of Computer Science, Friedrich-Alexander University Erlangen, Nürnberg, Germany

ARTICLE INFO

Article history:
Received 19 January 2015
Received in revised form
20 April 2015
Accepted 8 May 2015
Available online 22 May 2015

Keywords:
Android malware
Dynamic code loading
Dynamic script loading

ABSTRACT

In the work at hand, we first demonstrate that Android malware can bypass current automated analysis systems, including AV solutions, mobile sandboxes, and the Google Bouncer. A tool called *Sand-Finger* allowed us to fingerprint Android-based analysis systems. By analyzing the fingerprints of ten unique analysis environments from different vendors, we were able to find characteristics in which all tested environments differ from actual hardware. Depending on the availability of an analysis system, malware can either behave benignly or load malicious code dynamically at runtime. We also have investigated the widespread of dynamic code loading among benign and malicious apps, and found that malicious apps make use of this technique more often. About one third out of 14,885 malware samples we analyzed was found to dynamically load and execute code. To hide malicious code from analysis, it can be loaded from encrypted assets or via network connections. As we show, however, even dynamic scripts which call existing functions enable an attacker to execute arbitrary code. To demonstrate the effectiveness of both dynamic code and script loading, we create proof-of-concept malware that surpasses up-to-date malware scanners for Android and show that known samples can enter the Google Play Store by modifying them only slightly.

1. Introduction

The processing power of current smartphones already surpasses that of last decades personal computers. When they became a lot faster, our usage pattern quickly shifted away from solely placing phone calls. Today, we do not only store our family, friends, and business contacts on smartphones, but also check for emails, visit social networks, and access bank accounts to transfer money. Both, our private and business life, takes place on smartphones. By this, we entrust our personal data to them, including photos, movement profiles, and text messages.

As of today, the largest operating system for smartphones is Android with a market share of 81.9% (Gartner, Nov. 2013). This makes Android an ideal target for shady IT practices. Android versions on older phones do not get updated quickly, sometimes not at all. As a consequence, publicly available exploits can be targeted towards a large number of users for a long period of time. The *Android Market*, which is today rebranded to *Google Play Store*, has had a lot of malware uploaded in the past. Besides the Play Store, the open source nature of Android allows vendors to ship their devices with markets from other companies, like the *Amazon Appstore*, and also holds an option for users to install apps directly from the Internet.

* *Corresponding author*.
E-mail addresses: dominik.maier@fau.de (D. Maier), mykola.protsenko@fau.de (M. Protsenko), tilo.mueller@fau.de (T. Müller).
http://dx.doi.org/10.1016/j.cose.2015.05.001

1.1. Contributions

We investigated Android malware that behaves benignly in the presence of analysis, and maliciously otherwise. Automated static analysis scans all code that is locally present in cleartext, while dynamic analysis executes the code and observes its behavior. To surpass both analysis techniques, we exploit *split-personality* (Balzarotti et al., 2010) attacks that first split malware samples into benign and malicious parts, and then mislead security scanners by showing only benign behavior within analysis environments.

In detail, our contributions are as follows:

- We study the use of dynamic code execution techniques in malicious and benign apps by analyzing a set of 14,885 malicious and 22,032 benign apps. From the malicious set 36.4% of the samples use code loading, while from the benign set only 13.1% use code loading.
- As part of our work, also a fingerprinter for Android sandboxes has been developed called *Sand-Finger*. With Sand-Finger, we were able to gather information about ten Android sandboxes and AV scanners which are available today, such as *BitDefender* and the *Mobile Sandbox* (Spreitzenbarth et al., 2013).
- With the results of Sand-Finger, we prove that Android malware can easily gather information to surpass current AV engines by split-personality behavior. We provide proof-of-concept samples that use a combination of fingerprinting and dynamic code loading to enter the Google Play Store.
- We show that an attacker can, even in an environment that forbids to dynamically load compiled code, still run arbitrary statements by using scripting environments like Android's v8 JavaScript-VM to execute code that interacts with the Android OS. In a proof-of-concept, we show it is possible to run most statements in the Android API by combining dynamic script loading attacks with Java's reflection API.
- Finally, we give an overview about sandbox improvements and countermeasures that have been previously proposed to help prevent such attacks. We argue, however, why these improvements are difficult to implement on Android and why malware detection generally fails in practice.

To sum up, we show that split personality malware based on dynamic code loading is a real threat that can effectively surpasses all security mechanisms for Android, including sandboxes, the Google Bouncer and on-device AV scanners.

1.2. Related work

How to surpass virtualization-based sandboxes on x86 has been shown by Chen et al., in 2008 (Chen et al., 2008). A comprehensive security assessment of the Android platform has been given by Shabtei et al., in 2010 (Shabtai et al., 2010). The security of automated static analysis on Android, especially concerning virus scanners, has recently been evaluated in 2013, for example, by Rastogi et al. (Rastogi et al., 2013), Fedler et al. (Fedler et al., 2013a, b, c), and Protsenko and Müller (Protsenko and Müller, 2013). However, none of these approaches focus on bypassing dynamic analysis of automated Android sandboxes. The bytecode-based transformation system PANDORA (Protsenko and Müller, 2013), for example, complicates static analysis by applying random obfuscations to malware, but cannot surpass dynamic analysis as it does not change its behavior.

From an attackers point of view, semantics-preserving obfuscation that confuses static analysis is outdated due to the increasing use of Android sandboxes, like the Mobile Sandbox (Spreitzenbarth et al., 2013), and at the latest since the initial announcement of the Google Bouncer in 2012 (Lockheimer, Feb. 2012). A short time after the announcement of the Bouncer, it was proven that dynamic analysis cannot be entirely secure by Percoco and Schulte (Percoco and Schulte, 2012), as well as Oberheide and Miller (Oberheide and Miller, 2012). Most recently, in 2014, Poeplau et al. have shown that dynamic code loading can bypass the Google Bouncer (Poeplau et al., 2014). Poeplau et al. come to the conclusion that dynamic code loading is unsafe in general and must be secured on Android. However, obfuscation and packing techniques that bypass all Android sandboxes and AV scanners have not been investigated in a large-scale study yet.

The paper by Poeplau et al. specifically centers around dynamic code loading. They come to the conclusion that dynamic code loading is unsafe and should be secured on Android. They also propose a possible solution that we evaluate in this paper (Poeplau et al., 2014). The other evaluated solutions to secure dynamic code loading, targeted mainly towards native code, have been released by Fedler et al. (Fedler et al., 2013a, b, c). A comprehensive security assessment of the Android platform has been given by Shabtai et al. (Shabtai et al., 2010). Short time after the initial announcement of *Android Bouncer*, a new security layer on Android it was proven that it is not 100% secure. This conclusion was mainly underpinned by the work "Adventures in Bouncerland" from Percoco and Schulte (Percoco and Schulte, 2012) as well as the talk at SummerCon 2012 by Oberheide and Miller (Oberheide and Miller, 2012). They fingerprinted the Android Bouncer and used dynamic code loading to successfully submit malicious payloads to the Android Play Store.

Fingerprinters that track Android malware sandboxes to be used in split-personality malware have been proposed and their use against those sandboxes has been proven by different researchers (Maier et al., 2014; Petsas et al., 2014; Vidas and Christin, 2014). Luo et al. (Luo et al., 2011) have suggested attackers might exploit a WebView inside a previously prepared app. In addition to the Android world, research on how to evade VMs has been conducted on Windows for years, such as (Raffetseder et al., 2007). Loading malicious code dynamically as a scripting language has also been used by malware in the wild. Research on the "Flame" sample has been conducted and released by Bencsáth et al. (Bencsáth et al., 2012).

1.3. Outline

The remainder of this paper is structured as follows: In Sect. 2, we provide necessary background information for available anti-malware technologies on Android, such as AV scanners and sandboxes. In Sect. 3, we overview the widespread of

dynamic code execution techniques in malicious and benign apps in the wild. In Sect. 4, we describe our own approach to fingerprint analysis environments with the tool *Sand-Finger*. In Sect. 5, we show that Android malware can easily bypass automated security scanners by acting according to the principle of code splitting. In Sect. 6, we show how countermeasures that prohibit dynamic code loading can be tackled by dynamic script loading in the future. In Sect. 7, we discuss how this problem can again be tackled by looking at more sophisticated countermeasures. Finally, in Sect. 8, we conclude that dynamic code loading attacks yield an arms race on Android, or *a game of cat-and-mouse*.

2. Background information

We now provide background information about automated malware analysis for Android. In Sect. 2.1, we illustrate static analysis, pointing out some advantages and problems. In Sect. 2.2, we focus on dynamic analysis.

2.1. Static analysis

Static analysis is the art of analyzing code without actually running it. Static analysis tries to find out the control flow of an application in order to figure out its behavior (Moser et al., 2007). It is also often applied for finding programming faults and vulnerabilities in benign applications, and is furthermore used to detect malicious behavior without having to run an application. This paper uses the term static analysis for every analysis that is performed statically as opposed to running code dynamically. Note that static analysis is often understood as the task of *manually* analyzing malware, which is very time consuming and requires specially trained analysts. The principles discussed in this paper cannot defeat manual analysis entirely. However, as it is nearly impossible to perform manual analysis for all apps that are uploaded to the Play Store, we confine ourselves to automated analysis.

Currently, virus scanners on Android are not allowed to monitor running apps, but can only statically analyze them. Due to these restrictions, monitoring an app's behavior on real devices is close to impossible. Tests show that static virus scanners on Android can easily be evaded; proof has been released by various authors (Protsenko and Müller, 2013; Rastogi et al., 2013; Fedler et al., 2013a,b,c; Poeplau et al., 2014). For manual static analysis, sophisticated tools exist for both DEX bytecode and compiled ARM binaries. Contrary to that, considering automated static analysis on Android, native binaries are often black boxes for malware scanners since the tools focus on DEX bytecode. In native code, however, root exploits can open the doors for dangerous attacks.

2.2. Dynamic analysis

As a solution to the shortcomings in automated static analysis, running code in a sandboxed environment received attention. By running code instead of analyzing it, automated tools can trace what an app really does. The logging files can then be rated automatically, or be examined manually. Although approaches for running software on *bare-metal* exist (Kirat et al., 2011), dynamic analysis of mobile apps is almost exclusively done on virtualized environments. Virtual machines considerably simplify the design of sandboxes because (1) changes in a VM state can be monitored by the host system, and (2) a VM can be reset to its initial state after an analysis.

Sandboxes for desktop malware have existed for years, including the *CWSandbox* proposed by Willems et al. (Willems et al., 2007) as well as *Anubis* (Bayer et al., 2009). With the rise of smartphones, sandboxes for mobile systems were also proposed at that time (Becher & Freiling; Becher & Hund, may 2008). After the introduction of touch-centric OSs for smartphones, in particular iOS and Android, old smartphone systems quickly declined in sales and researchers started to adopt sandboxes for the new platforms. On the one hand, the openness of Android leaves the possibility to spread malware more easily, but on the other hand, thanks to this openness, it has proven to be easier to implement a sandbox for Android than for iOS (Bläsing et al., 2010). Recent sandbox technologies are mixing static and dynamic analysis for the best results. As shown in our comparison of sandboxes in Sect. 4.2, many sandboxes also use AV vendors to detect Android malware.

3. Dynamic code loading in the wild

The widespread of dynamic code loading techniques was first studied by Poeplau et al. (Poeplau et al., 2014) for benign apps. They outlined five mechanisms of loading and executing additional code, previously not included in the original APK. These mechanisms are class loaders, package contexts, the use of Runtime class, increase of the code basis through native code, and installation of additional APKs. They have statically analyzed over 1700 benign apps and found one third of them to be using one of these code loading techniques. Here, we have conducted a similar study that is limited to the most important mechanisms class loaders and the use of Runtime class, but compares a larger dataset of 14,885 malware samples and 22,032 benign apps.

For our evaluation, we implemented an analysis tool based on the Soot framework (Vallée-Rai et al.) which is able to directly input from.apk files and perform various custom analyses on the intermediate representation without requiring the source code of an app. Dynamic code execution techniques considered in this study, as well as our approach to their detection are briefly explained below:

- *Class Loaders* allow the app to dynamically load additional classes from various file formats and locations. We recognize an app to be using this technique if it calls a method with the name 'loadClass' of some class with the name ending with 'ClassLoader', e.g., DexClassLoader.
- *exec()-Method of the Runtime class* allows execution of an arbitrary binary file. Detecting this feature can be done by searching the bytecode instructions of an app for the invocation of the exec() method from the Runtime class.

The results of this investigation are presented in Fig. 1. Summarizing, 36.4% of the malicous samples and 13.1% of the benign apps were detected to be using dynamic code loading. Note that these ratios are lower bounds because we searched

	Malicious	Benign
Samples total	14885	22032
Analysis Failed	2350	117
Processed Successfully	12535	21915
Class loading	1118 (8.9%)	1546 (7.1%)
`Runtime` class	3769 (30.1%)	1675 (7.6%)
Both	329 (2.6%)	358 (1.6%)
Either	4558 (36.4%)	2863 (13.1%)

Fig. 1 – Widespread of dynamic code loading in Android applications.

for the most frequently used code loading mechanisms but may have missed apps that use another technique. Also note that direct processing of Android bytecode in Soot is a new feature under active development which is not always stable. Hence, the number of analysis failures is shown in the third row of the table.

The results are consistent with those obtained by Poeplau et al. and show that dynamic code loading is not a rare occurrence, which indicates the relevance of the problems brought to discussion in this paper. Furthermore, one can see that malicious apps make use of dynamic code execution generally more frequently than benign ones.

4. Fingerprinting analysis environments

In this section, we introduce our fingerprinting tool *Sand-Finger* (Sect. 4.1), and give a comparative overview of available sandboxes (Sect. 4.2).

4.1. *The sandbox fingerprinter*

Smartphones are usually embedded devices with built-in hardware that cannot be replaced and, as opposed to desktop PCs, they are aware of their specific hardware environment. Many different kernel and Android versions exist with various features for different hardware. Additionally, smartphones are actively being used and are not run as headless servers. In other words, users install and start different apps, interact with the GUI, lock and unlock the screen, move the device, and trigger numerous other system events. All this device information, including the specific hardware, location data, and user generated input, can be exploited to differentiate systems, which are used by humans, from automated analysis environments. Hence, this information can be exploited to detect dynamic analysis environments.

During the development of the Sand-Finger fingerprinter, we collected as much data as possible from sandboxed environments and submitted it to a server for evaluation. Building a VM close to actual hardware is a sophisticated task due to the large number of sensors, e.g., accelerometer, gyroscope, camera, microphone, and GPS. In short, we could not find any sandbox with hardware that is entirely indistinguishable from real hardware. In addition to hardware, malware can read settings and connections to find out whether it is running on a real device or not. For example, constraints like "the device needs at least *n* saved WiFi-networks", "the device must have a paired Bluetooth device, the device must not be connected via cable", "the device IP must not match a specific IP range", and "the device must have at least *n* MB of free space", can easily be put in place and require dynamic analysis tools to emulate realistic environments.

Note that the examples given above are not far-fetched, because at the moment no emulator we analyzed has an active or saved WiFi connection. Also note that detection rates with a small number of false positives do not prevent malware from being spread. To get accurate fingerprint results, additional oddities can be taken into account. For example, the default Android emulator does not forward the ICMP protocol used by ping. Whenever an obvious problem, like the empty WiFi list or missing ICMP forwarding, gets fixed, malware authors will come up with new ideas. Instead, static analysis must be used to reach high code coverage and to skip the fingerprinting part of malicious apps (Spreitzenbarth, 2013). However, according to our results, none of the tested analysis environments can fool simple fingerprinting and reach high control flow coverage.

The app that we developed for this work, namely Sand-Finger, sends an automatically generated JSON-encoded fingerprint to a remote web service via HTTPS. JSON *(JavaScript Object Notation)* is a human-readable data format for which many parsers exist. HTTPS is used to prevent the filtration of fingerprint data. While performing our tests, we noticed that some requests were filtered and never reached our server. Some of these sandboxes, however, could then be assessed without encryption.

When executed in a sandbox, the fingerprinter gathers information available on the virtual device. Around 100 unique values are extracted, depending on the platform version. The information includes static properties like the device "brand", "model", "hardware", "host name", "http agent", and a value named "fingerprint". Information about the underlying Linux system is extracted as well, such as "Kernel version", the "user", and the "CPU/ABI". On top of this static information, a lot of device settings are gathered, such as the timezone, the list of WiFi connections, and whether debugging is enabled. Additional data is gathered from the surroundings, namely the current phone's cell network status and neighboring cell info. Our analysis also checks if the device has been rooted by searching for the *su* binary. Moreover, it lists the directories of the SD card and the system's bin-folder and transmits all available timestamps.

According to our tests, it is very useful to know how long a system has already been running, since analysis systems are often reset. We also send a *watermark* back to our server, which is a constant string that is specified at the compile time of Sand-Finger to identify the origin of each test run. Thanks to this, the redistribution of malware samples between sandboxes and AV vendors can be traced. For example, the Mobile Sandbox passes on suspicious samples to companies

like *BitDefender* and *Trend Micro*, if a sample was not marked as private.

Last but not least, two pings to google.com and our own server are performed on the command line. Since the *QEMU* VM, in its default mode for the Android emulator, does not forward packages other than UDP and TCP, an app is presumably inside the default emulator if no ping reply is received. Hence, the values returned by ping are also added to our JSON-fingerprint. After a complete fingerprint is received by the server, it is stored inside a database together with the server's timestamp and the sender's IP address. Sand-Finger can also embody malware, that combines dynamic code loading against static analysis with a split-personality behavior, making it invisible to automated static and dynamic testing. The client listens for server responses, and only if an expected response is received, does Sand-Finger extract its payload and run it using System.load().

4.2. *Comparing Android sandboxes*

In this section, we give an overview about available Android sandboxes and AV scanners, and compare these analysis environments based on the results of our fingerprinter Sand-Finger. Anti-virus companies often receive samples from sandboxes; for example, according to the identification of our watermarks, the Mobile Sandbox forwards samples to several online AV scanners, including BitDefender and Trend Micro. The IP of one fingerprint we received for the watermarked Sand-Finger sample that was submitted to the Mobile Sandbox belongs to the anti-virus company BitDefender, while another was sent from a system with the model *TrendMicroMarsAndroidSandbox*. Contrary to that, the Google Bouncer does not forward submitted samples to AV scanners. Moreover, the Google Bouncer does not seem to connect to any internet service, including our web server waiting for fingerprints. Consequently, we could not reliably fingerprint the Google Bouncer (but we were still able to surpass it with simple split-personality malware, as explained in Sect. 5.2).

As the JSON file generated by Sand-Finger has over 100 entries to identify a system, we focused on values in which all analysis environments greatly differ from real hardware. Altogether, we found about ten of such values, as listed in Fig. 2. The hardware we used as a reference value is the *Nexus* 5 from Google, but we confirmed our results on other hardware, too. As a second reference value, we used the official emulator of the Android SDK. The meaning of the columns in Fig. 2 is:

- *Android Version*: The version number of Android.
- *CPU Architecture*: The actual CPU architecture. In some cases, the CPU/ABI differs from the *os.arch* type in Java, so the latter was used.
- *Resembles Hardware*: Some sandboxes try to adapt the virtualization environment to resemble real devices, mostly Google Nexus. However, others just use stock values or something completely different.
- *Successful Ping*: The official Android emulator cannot forward pings, only UDP and TCP packets are routed.
- *Correct Timestamp*: The timestamp in virtualized environments is oftentimes inaccurate, but accurate timestamps can be received from the Internet. This column lists if the timestamp is close to correct.

	Sandbox	Android Version	CPU Architecture	Resembles Hardware	Successful Ping	Correct Timestamp	Hardware not Goldfish	WiFi Connected	Cell Info	UpTime >10 min.	Consistent	Untraceable
	Google Nexus 5	4.4.2	armv7l	✓	✓	✓	✓	✓	✓	✓	✓	-
	SDK Emulator (x86)	4.4.2	i686	X	X	✓	X	X	✓	X	✓	-
1	Andrubis	2.3.4	armv5tejl	X	X	X	X	X	X	X	✓	-
2	BitDefender	4.3	i686	X	X	✓	X	X	✓	X	✓	X
3	ForeSafe	2.3.4	armv5tejl	X	X	✓	X	X	X	✓	✓	✓
4	Joe Sandbox Mobile	4.2.1	i686	✓	✓	X	✓	X	X	X	✓	✓
5	Mobile Sandbox	4.1.2	armv7l	✓	X	✓	✓	X	X	X	✓	-
6	SandDroid	2.3.4	armv5tejl	X	X	X	X	X	X	X	✓	-
7	TraceDroid	2.3.4	armv5tejl	X	X	✓	X	X	X	X	✓	X
8	Trend Micro	4.1.1	armv7l	X	X	X	X	X	X	X	✓	X
9	Unknown ARM	4.1.2	armv7l	✓	X	X	X	X	X	X	X	✓
10	Unknown x86	4.0.4	i686	✓	X	X	✓	X	X	X	X	✓

Fig. 2 – Fingerprinting ten Android sandboxes with Sand-Finger. The eleven most revealing values (out of over 100) are listed here as examples.

- *Hardware not Goldfish*: The virtual hardware of the Android emulator is called *Goldfish*. This column lists if the tested environment uses a different hardware.
- *WiFi Connected*: A real device is usually connected to the Internet either via cell-tower or WiFi, otherwise the fingerprint could not reach our server. Sandboxes are often connected via cables instead.
- *Cell Info*: If the device is a real phone, it has some kind of cell info describing the current mobile connection. Sandboxes often omit this info.
- *UpTime >10 min*: As a normal phone or tablet runs the whole day, the probability of a device running longer than 10 min is high. Sandboxes are often reset after each analysis.
- *Consistent*: A consistent fingerprint from a real device does not show two different Android versions, or two different CPU architectures.
- *Untraceable*: A sandbox that gives away its own location or its IP address in its fingerprint is traceable; otherwise it is untraceable.

In the following, we discuss the results of ten fingerprinted sandboxes and AV scanners in alphabetical order: *Andrubis*, *BitDefender*, *ForeSafe*, *Joe Sandbox Mobile*, *Mobile Sandbox*, *SandDroid*, *TraceDroid*, *Trend Micro*, and two systems from unknown vendors. The unknown systems stem from the redistribution of malware samples among sandboxes. We received several fingerprints from these sandboxes, but could not identify their operator.

4.2.1. *Andrubis*

This is the mobile version of *Anubis*, a well-known sandbox for desktop PCs (Bayer et al., 2009). Andrubis performs static as well as dynamic analyses and allows uploading APK files. It also offers an Android app to submit APK files. A problem about the APK files, however, is the size restriction of seven MB (Lindorfer, Jun. 2012). Andrubis has an extensive feature set, including support of the native code loading. Like many other sandboxes, it is based on open source projects like *DroidBox*, *TaintDroid*, and *Androguard* (Desnos). The Andrubis sandbox, despite its usefulness when it comes to detection of malware, can easily be fingerprinted. Its environment is similar to the official emulator, and it even uses test keys (meaning it is self-signed). Another drawback is, that it still runs on Android version 2.3.4, probably because of its dependency on the popular analysis software TaintDroid (Enck et al., 2010). As on most sandboxes, also on Andrubis, a ping did not reach its destination. The static IP address used by Andrubis ends at the Faculty of Informatics of the TU Vienna. Regarding this sandbox, we noticed strange behavior when an app closes: Andrubis just taps on the desktop wildly, launching the music app or executing a Google login, for example.

4.2.2. *BitDefender*

In the few weeks of our analysis, while we were working on the fingerprinter, the BitDefender sandbox apparently was updated. First, the VM was inconsistent, actually being a 4.1 Android that claims to be 4.4. Then the system changed, and since then consistently shows Android 4.3. This makes it the most recent Android version among the group of tested sandboxes. However, BitDefender is not the stealthiest environment because values like "Batmobile" for manufacturer and "Batwing" for the product can easily be fingerprinted. But note that as long as nobody fingerprints these values, they are more secure than the stock names from the Android emulator, because it is impossible to white-list all Android devices. Interestingly, BitDefender was the only sandbox that contained cell information to emulate a real phone. The static IP address used by this sandbox is registered for BitDefender.

4.2.3. *ForeSafe*

The ForeSafe sandbox not only offers a web upload but also a mobile security app for Android. After analysis, ForSafe shows an animation of the different UI views it passed while testing. It is not the only Sandbox that captures screenshots, but the only one with actually moving screenshots (ForSafe, 2013). ForeSafe uses Android 2.3.4 and hence, probably builds on TaintDroid. Despite heavy fingerprinting and the possibility to execute native code, Sand-Finger was flagged as benign by this scanner. ForeSafe executed the sample a few times and was the only scanner with an uptime exceeding 10 min. This is important as malware could simply wait longer than 10 min before showing malicious behavior.

4.2.4. *Joe sandbox mobile*

This is a commercial sandbox which has a free version as well. Just like the sandboxes above, it is possible to upload APK files for automated testing. According to our tests, this sandbox performs very extensive tests, combining static and dynamic analysis, and it even detects some anti-VM techniques in malware. Roughly speaking, it is the most exotic sandbox in our test set. First, we found out that Joe Sandbox uses the anonymous Tor network as proxy and is thereby untraceable. Second, it is the only sandbox that does not build upon the official Android emulator, but uses an x86 build of Android for the *Asus Eee PC*. In addition to the x86/ABI, Joe Sandbox specifies "armeabi" as "CPU2/ABI", meaning that Intel's *libhoudini.so* is probably installed to run binary translated ARM code. Most likely, Joe Sandbox Mobile is running on VMware; the build settings, however, are adapted to mimic a Galaxy Nexus. As it is based on a different virtualization technology, pings can reach their destination. Moreover, the build is rooted. The sandbox itself was generally hard to trick, e.g., some anti-virtualization techniques are detected.

4.2.5. *Mobile sandbox*

Another advanced sandbox with many features in the making, like machine learning algorithms, is the Mobile Sandbox. It offers a web platform to submit malware samples and is based on a modified version of DroidBox for dynamic analysis. It also applies static analysis and monitors usage of native libraries as well as network traffic. The camouflage of Mobile Sandbox is the most advanced within our test set. Just by looking at its properties, it is indistinguishable from a Samsung *Nexus S*, even replacing the Goldfish hardware string with a real device string. Moreover, a telephone number was given, and it has a more recent Android version than most sandboxes we tested. Interestingly, this sandbox sent the most fingerprints over a time period of 20 min, meaning it restarted the app over and over, probably trying to maximize

code coverage. Mobile Sandbox also applies some machine learning algorithm to the results and sent data. However, it shares some of the flaws all Android emulator-based sandboxes have, like the missing ping and a static IP address.

4.2.6. *SandDroid*

SandDroid is a Chinese online analysis tool (Botnet Research Team, 2014). This sandbox performs detailed static analysis which it presents graphically to the user. It is powered by Androguard and DroidBox, and allows to upload APK files of up to 20 MB. Besides its rich static analysis, it features a dynamic analysis and also gives a security rating. The fingerprint of Sand-Finger never reached our web server, so HTTP is probably sandboxed, too. Requests returned "302:Found". However after uploading a non-HTTPS version of Sand-Finger, the fingerprint results could simply be copied from SandDroid's traffic capturing. The timestamp inside the sandbox is wrong. According to the low Android version, SandDroid probably uses TaintDroid.

4.2.7. *TraceDroid*

The *TraceDroid Analysis Platform (TAP)* is the only sandbox that rarely performs static analysis and does not rate the results (Van der Veen, Aug. 2013). Instead, it packs all results in an archive that can then be downloaded from a web service. The data is useful, especially the control flow graphs. Regarding the fingerprint, it is interesting that many log files are stored on the SD card. TraceDroid uses *Monkey Executor* to generate random user input. The sandbox is easily traceable because the IP of the sandbox is equal to the static IP of the TraceDroid web server located at the VU University Amsterdam.

4.2.8. *Trend Micro*

Another anti-virus scanner that is used by the Mobile Sandbox and sends fingerprints to our web server is Trend Micro. The fingerprinted sandbox of Trend Micro has the model string *TrendMicroMarsAndroidSandbox* and its IP address is located in Japan (where this company is located). It is a sandbox running a rather recent Android version, namely 4.1., and probably builds upon DroidBox. Interestingly, the user name is always "mars" and the SD card contains a PCAP-file and a file named *mars_conf.txt*. Hence, this environment can be fingerprinted with static strings.

4.2.9. *Unknown ARM and x86*

We received fingerprints from more AV engines that we could not match to a specific operator yet. These fingerprints were watermarked with the Sand-Finger samples that we submitted to the Mobile Sandbox. The Mobile Sandbox submits samples to more AV companies, including *ClamAV*, *Kaspersky*, *Sophos*, and others. The yet unknown ARM-based sandbox seems to be built on DroidBox and its IP address is located in China, pointing to a Chinese AV company. The sandbox is rooted but has no super user APK installed. While the fingerprint states the device is a *google/occam/mako*, thus mimicking the Google *Nexus* 4, the Android version in reality is 4.1.2 and is based on the Goldfish hardware. The unknown x86 sandbox is similar and also mimics a Nexus device with a higher fingerprint than the platform version. It claims to be the *google/soju/crespo*, i.e., a Nexus S, but only runs Android 4.0.4. Moreover, it claims to be an "armeabi-v7a" architecture although it is actually x86-based. This implies that, just as Joe Sandbox Mobile, this emulator has the ARM-to-x86 binary translator *libhoudini.so* installed. The x86 sandbox runs on a dedicated server in France.

4.3. *Summary*

All of the fingerprinted sandboxes in our test set have specific elements in common: (1) None of the analysis environments could send an ICMP ping packet except Joe Sandbox Mobile. (2) No sandbox has been up for more than 10 min except the ForeSafe sandbox, which has been up for 15 min. This implies that no sandbox ran our Sand-Finger sample for more than 15 min (3) None of the tested sandboxes returned a WiFi connection or stored any WiFi SSIDs. Only one sandbox was virtually connected to a cellphone network, but real devices can send back their fingerprints only if either a WiFi or a cellphone network is connected. (4) Most of the VMs are simply based on the emulator hardware Goldfish. (5) The timestamp was often incorrect. The timestamp, however, is clearly not the best sandbox detection method.

Considering current versions of the sandboxes we tested, we can conclude that using the uptime of a system in combination with its hardware string and a list of connected networks is a reliable indicator to differentiate between virtualized environments and real devices. The false positive rate of this combination would already be minimal, possibly concerning only devices that were recently switched on. Some of these indicators can be fixed by sandboxes in the future, but in turn the indicators used by split-personality malware can be changed, as well. Hence, creating an Android sandbox that cannot be distinguished easily from real devices remains an interesting research topic for malware analysis.

5. Defeating analysis environments

Insights from our fingerprinting tool Sand-Finger can be deployed to build split-personality malware that behaves maliciously on real devices, but gets classified as benign in analysis environments. To defeat sandboxes in general, we focus on a technique based on split-personality behavior. These attacks divide malware samples into a benign and malicious part, such that the malicious part is hidden from analysis by packing, encrypting or outsourcing of the code. As a consequence, the malware sample outwardly appears benign and hence, misleads the AV engine. Only when running on real devices, does split-personality malware show its true behavior. These methods are especially effective, as they are hiding from both static and dynamic analysis. Malicious code segments are split in such a way that a sandbox is not aware of critical code.

Note that split-personality attacks are especially successful against restrictive operating systems like Android. On an unrooted Android phone, it is impossible for an app to scan another app during runtime, preventing on-device AV scanners from applying heuristics after a sample has unpacked itself. Contrary to that, proper virus scanners for Windows desktop PCs cannot be tricked as easily, as they run in a

privileged mode and monitor the address space of third party apps at runtime. Indeed, many apps exist for Android that suggest effective AV scanning, but technically these apps can only scan a sample statically before runtime.

5.1. *Collusion attacks defeat VirusTotal scanners*

Automated analysis usually consists of a static and a dynamic part. In the static phase, sandboxes first identify granted permissions in the manifest file, scan through constants, embedded URLs, and possibly library calls. Afterwards, they execute the code, oftentimes tapping on the screen more or less randomly, and finally rate the sample. Commonly, Android permissions play a central role in this rating. By doing so, sandboxes can classify ordinary apps that come as single APK files. Malware, however, can also be split into two complementary apps.

Due to the way Android's permission system works, an app can access resources with the help of another app, in such a way that it does not need to have all permissions declared in its own manifest. Those attacks are called *collusion attacks* (Marforio et al., 2012): Assuming the app with the desired permissions has already been trusted by the user on installation, another app can send the requests to perform a certain action by IPC mechanisms, like sockets or even covert channels. On a system as complex as Android, many covert channels can be found, for example shared settings. Even using the volume as a 1-bit channel has been proposed (Bugiel et al., 2012). However, social engineering techniques have to be used to get a user to download the complementary app. But it is a common practice to provide additional app packages, like game levels, plug-ins, and more.

The collusion attack is effective against automated static and dynamic analysis alike, because automated analysis looks at only one app at a time. Also note that the collusion attack is currently the only method, besides root exploits, that can trick the analysis of permissions. To evaluate this approach, we used the open source malware *AndroRAT (Android Remote Administration Tool)*. This software connects to a server which then controls the whole Android system remotely. Many AV scanners detect the installation of AndroRAT as malware, namely 15/50 AV solutions on *VirusTotal*. Microsoft's Windows Defender even stopped the compilation of AndroRAT from source.

As part of this work, we created three variants of AndroRAT, called *AndroRAT-Welcome*, *AndroRAT-Pandora* and *AndroRAT-Split*. AndroRAT-Welcome simply includes an anti-automation screen that must be touched by the user before the app continues. AndroRAT-Pandora is a obfuscated version of AndroRAT that we created with the transformation framework PANDORA (Protsenko and Müller, 2013) to confuse static analysis. AndroRAT-Split finally splits the app into two and performs a collusion attack as described above. Technically, we kept the main service class in one APK and added the activity classes to a second app. Thanks to Android's IPC mechanisms, this approach was straightforward to implement.

While the detection rate of the original AndroRAT sample on VirusTotal is 15/50, the detection rate of AndroRAT-Welcome declined to 9/15. Thus, six AV engines were either not able to bypass the welcome screen, or detecting the original version was based on static signatures, like a hash sum of the AndroRAT APK. While these detection rates are already fairly low, our goal was to decrease them further. The detection rate of AndroRAT-Pandora declined to 4/15, probably defeating static analysis systems in the first place. The detection rate of AndroRAT-Split finally declined to 0/15 for both of the separated apps, since none of them shows malicious behavior without the other.

We also tested the collusion attack against our test set of dynamic sandboxes. When analyzing with TraceDroid, a sandbox without static analysis, none of the malicious methods was detected. On sandboxes that combine static and dynamic analysis, however, only the activity app was rated as benign. This is plausible, as without the service class, the app has no harmful code. The service app was generally considered less harmful but still precarious. Hence, we could not surpass all sandboxes with a simple collusion attack. More advanced variants of AndroRAT-Split would be required that either split up the permissions into more apps, or combine the collusion attack with fingerprinting and dynamic code loading, as described before.

5.2. *Dynamic code loading defeats the Google Bouncer*

In this section, we describe how split-personality malware can be combined with dynamic code loading to enter the Google Play Store. To enter the Play Store, an Android app must pass the Google Bouncer, which is the most prominent example of dynamic analysis for Android today.

With respect to dynamic code loading, Android imposes minimal restrictions inside an app (whereas Apple, for example, tries to restrict dynamic code loading). From an AV engine's point of view, the problem is that dynamic code loading is a common practice in many benign Android apps, too. Just like on Windows, Android apps are free to load whatever they please at runtime. On Windows, however, effective malware scanners can be installed that monitor other apps at runtime.

As opposed to this, on Android every app runs its own Dalvik VM instance that cannot be accessed by other apps. While this is, on the one hand, secure because apps are not allowed to alter other apps in a malicious way, *apps can do anything they have permissions for*. Once an Android malware sample defeats the Google Bouncer and enters the Play Store, e.g., based on split-personality behavior, there is no effective authority that monitors an app while running on real devices.

The possibilities for virus scanners to counteract threats on Android are therefore astonishingly small. Static analysis can understand every instruction but doesn't see the important parts as they are loaded later and dynamic analysis could easily analyze the code loaded at runtime, except it is not loaded at all in this case (Petsas et al., 2014; Maier et al., 2014; Vidas and Christin, 2014). On Windows, this kind of attack would prove impossible thanks to anti-malware solutions that monitor running apps. It's clear to see the security against dynamic code loading is currently insufficient on Android. Dynamic code can be downloaded from the Internet, copied from SD-card, loaded from other app's packages or even ship hidden in the own app's resources (Fedler et al., 2013a,b,c).

Apart from static analysis, that will not be able to find the dynamically loaded code, loading the code only while not being watched will also protect the malware from dynamic analysis. So combining dynamic code loading with one of the split personality methods discussed in Sect. 4 results in a very well hidden malware. So even a thoroughly malware-scanned app can do anything as the scanner stops watching (Poeplau et al., 2014).

From a technical point of view, dynamically loaded code in Android can either be a DEX file or a native library. It can be downloaded from the Internet, copied from SD card, loaded from other apps packages or be shipped encrypted or packed in the apps resources (Fedler et al., 2013a,b,c). Apart from static analysis, which is naturally not able to find dynamically loaded code, dynamic analysis can be defeated when it is combined with fingerprinting techniques. Even a thoroughly scanned app can do anything after the scanner stops watching (Poeplau et al., 2014).

As we revealed during our analysis in Sect. 4, it is currently impossible to fingerprint the Google Bouncer because it blocks any Internet traffic to custom servers (or does not execute uploaded APKs at all). This, however, has been different in the past. Oberheide and Miller were able to fingerprint the Google Bouncer in 2012 (Oberheide and Miller, 2012). For example, they found out that the Google Bouncer is based on the Android emulator as it creates the file */sys/qemu_trace* during boot. As part of this work, the Google Bouncer was revisited. Since most devices are regularly connected to the Internet today, the missing Internet connection itself can be used as a fingerprint. That is, malware can disable malicious code on a missing Internet connection to evade dynamic analysis of the Google Bouncer. While cutting the Internet connection on a sandbox hinders fingerprinting, there is no way of executing malware that relies on connectivity to a command and control server.

For our tests, we equipped an otherwise benign app with a root exploit that is executed as soon as a certain command is received from a C&C server which is regularly contacted. For our first test, we included the root exploit as a plain native ARM binary into the app. Containing an executable root exploit, the app should be rejected based on static analyses of the Google Bouncer and not become available for download. As we expected, this variant was indeed classified as malicious and got rejected by the Bouncer soon after upload. As a consequence, the author's Google Play account was banned. Unfortunately, Google managed to link the banned account to all test accounts, although those were issued with fake identities, such that all of four developer accounts were closed at once. It is worth mentioning that the rejected sample was neither blacklisted by *Verify Apps* nor uploaded to the Google-owned *VirusTotal*, despite its being banned from the store.

Surprisingly, to confuse static analyses by Google's Bouncer, it turned out to be sufficient to zip the root exploit. Contrary to our own expectations, it was neither necessary to encrypt the root exploit, nor to dynamically download it from the C&C server. The Google Bouncer never contacted our server during its analysis, but the malicious APK became available for download on the Play Store a few hours after submission. To verify our approach, we downloaded the binary to a vulnerable Samsung Galaxy S3 device and ran the exploit. Note that, as our server always declined execution requests from unknown clients, there was no danger for Android users at any time. Only when the server permitted execution was the binary unpacked and executed. Even if the bouncer initially didn't run the binary but waited for the app to get a certain number of downloads, it had no way to detect our malware, so we took our sample from the Play Store within 24 h after all our tests had been applied, and we did not count any download of the app other than ours.

To sum up, malicious apps hiding a root exploit can easily surpass widespread security checks for Android. Malware without root exploits, however, would still require declaring its permissions in the manifest file. However it can pair this method with a collusion attack.

6. Dynamic script loading attacks

In an environment where loading binaries is prohibited, malware authors can still find an attack surface for dynamic code loading. Shipping a full-fledged scripting environment with an app is one of them. This method has already been applied by Windows malware like "Flame" (Bencsáth et al., 2012). Scripts can execute certain functions inside the binary and considering the spread of scripting environments in benign apps, flagging an app simply as malicious due to its use of a scripting host is no solution. Above that, unknown interpreters might be hard to track for automated analysis.

6.1. *Taming dynamic code loading*

Today the Android OS lacks a way to prevent attacks based on dynamic code loading. However, several countermeasures of this type have been proposed in the past, discussed in this section, that could make more advanced attacks necessary in the future.

Fedler et al. also proposed several methods to secure the Android platform against the execution of malicious native code, especially root exploits (Fedler et al., 2013a,b,c): An executable bit for Javas *System.load()* and *System.loadLibrary()* methods, a permission based approach where execution is granted per app, and the ability to load libraries only from specified paths. Above that, Fedler et al. proposed an antivirus API for Android that allows certified scanners to monitor the file system and third party apps on demand (Fedler et al., 2013a,b,c). For example, live change monitoring on the file system becomes possible similar to AV solutions on Windows. This API, if it were included in standard Android, could also help to mitigate the risks of dynamic code loading.

A more sophisticated approach towards taming dynamic code loading, regarding both native code and bytecode, has been presented by Poeplau et al. (Poeplau et al., 2014). Poeplau et al. suggest disallowing the execution of unsigned code entirely. They propose a patch to the Android system that allows multiple signing authorities and hence fits the Android ecosystem such that other companies, apart from Google, can still offer software in their own stores. If code is always required to be signed, so that only trusted code can be executed, on-device AV scanners would no longer be needed.

Revising and signing all code of an app store, however, is obviously a bottleneck.

6.2. Proof-of-concept implementation

In this section we provide a proof of concept, building on top of the V8 JavaScript Engine that is preinstalled on all Android systems. While it is possible to use V8 binaries directly, we did not want to load a native library, such that we took a detour to get to the interpreter right out of Java code. To this end, we leveraged Android's WebView, an UI element on Android that can show webpages and run JavaScript code. As it is possible to inject callbacks into the JavaScript code, JavaScript can call back to our app's code. This use of the WebView has indeed been noted before (Chin and Wagner, 2013; Luo et al., 2011; Neugschwandtner et al., 2013), but as of now, no usages as obfuscation method have been observed. To not be bound to preprogrammed code, we also show the use of reflection to call Android APIs out of JavaScript.

For our needs, the WebView had to be set up invisible and with JavaScript enabled. Scripts are disabled by default for security reasons, to keep developers from accidentally introduce scripting attacks to their apps. We then set our class as callback to be able to call it from our JavaScript code. The methods we want to be able to call from JavaScript need to be annotated with *@JavascriptInterface*. To make this technique a proper obfuscation method, we added fingerprinting from Sect. 4 to the app and made the app to only execute when being run on an actual phone (Maier et al., 2014; Petsas et al., 2014; Vidas and Christin, 2014). After the setup, we could successfully call pre-prepared methods in our classes from a webpage that has been loaded from the web.

Listing 1. Get object from "heap" and call function via reflection

```
1 //Get an object from the map
2 //call function
3 //store return to the map again
4 Object obj = heap.get(objStr);
5 cls = obj.getClass();
6 Method method = cls.getMethod(name, paramTypes);
7 heap.put(var,method.invoke(obj, paramInstances));
```

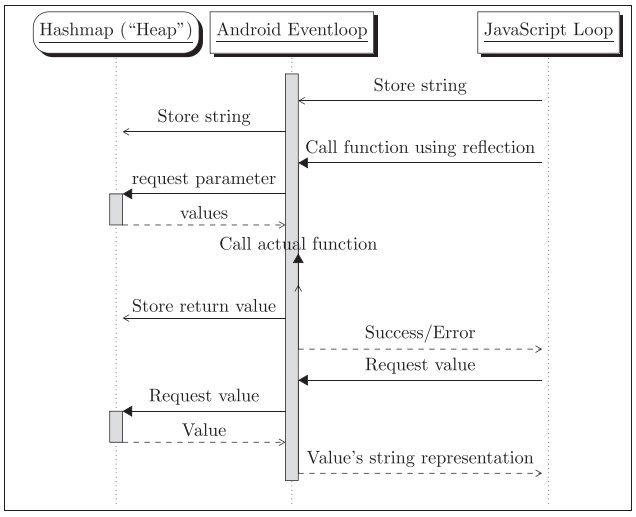

Fig. 3 – Interaction between JavaScript and the app.

reference for following function calls. Thanks to this virtual heap, we can call any Java methods with any parameters by simply storing them in the map and using the strings to specify them. The use of reflection can be seen in Listing 1. Our wrapper function takes care of mapping the given strings to objects. However, as we need to be able to read actual return values for conditionals and loops, we implemented an additional method that returns a string representation of the value to a given key inside the hashmap. To pass initial parameters, we added functions to store strings into the hashmap. The high-level overview over the interactions is shown in Fig. 3.

With only these three methods, calling a function using reflection, storing a string and getting a string back, a large part of the Android functions can be used. For convenience, we added methods to call constructors, get field values, pass integer values and arrays back and forth and free unused hashmap entries but all those could have been implemented in JavaScript as well. Additionally, before loading any JavaScript, we filled the hashmap with one value, namely the activity's context. A context in Android is needed for many API-calls in Android. For it we chose the key ctx. A lot of work has been performed on Java reflection, but by now it is still only possible to analyze reflection by combining static and dynamic analysis (Bodden et al., 2011; Hao et al., 2013). Using

Of course our problem is, especially if we want to use the Android API, that we need to use Java objects but cannot return or construct them inside the WebView. Note that this problem might not occur using alternative interpreters like Rhino.[1] To be able to use the Java VM anyway, we wrapped the Java reflection API once more: instead of returning objects, we store them into a hashmap that maps a string to an object. The key can be specified from JavaScript and can then be used as

[1] url https://developer.mozilla.org/de/docs/Rhino.

this setup, we were able to periodically send SMS containing the last known GPS position out of JavaScript using the code seen in Listing 3 inside JavaScript's setInterval() function. As comparison, the pure Java code to our JavaScript example is given in Listing 2. As to our expectation, none of the tested sandbox tools could spot this attack.

Listing 2. Java code to send an SMS

```
SmsManager.getDefault() .sendTextMessage(phoneNumber, null, "EvlSms:
    " + location.substring(0, location.length() − 40), null, null);
```

Listing 3. JavaScript code to send an SMS via reflection

```
// Put a string into the hashmap
app.createString('phoneNumber', NUMBER);
app.createString('text', SMSTEXT);
// call static method and store return as smsManager
app.callFunction('smsManager',
'android.telephony.SmsManager',
'getDefault', '[]');
// a class name as param instead of a key stands for null
app.callMethod('void', 'smsManager', 'sendTextMessage',
    '["phoneNumber", "java.lang.String", "text",
    "android.app.PendingIntent", "android.app.PendingIntent"]'));
```

7. Countermeasures

As seen in the previous section, prohibiting dynamic code loading on Android may not solve the problem of split-personality malware but yield an arms race where the attacker's next step would be to use dynamic script loading. In this section, we now discuss other countermeasures that have been proposed against the problem of split-personality behavior in the literature.

7.1. *Machine learning*

Instead of playing the ongoing game of cat-and-mouse between AV scanners and malware authors, many researchers propose using machine learning and anomaly detection algorithms (Shabtai et al., 2012; Arp et al., 2014). In theory, a well-trained machine learning algorithm not only identifies known malware samples, or those matching certain heuristics, but also yields to correct results for yet unknown malware. As an advantage, machine learning algorithms evolve over time and can learn from a few known malware samples that were analyzed manually. With a growing set of analyzed apps, the detection rate of machine learning becomes more accurate. Hence, this approach is particularly suitable for sandboxes, and some of the sandboxes we tested already started to apply machine learning algorithms, like the Mobile Sandbox.

7.2. *Improving sandboxes*

Many ideas that are used in both mobile malware and mobile sandboxes, have been found on desktop PCs before. For the case of desktop PCs, Kirat et al. proposed to run malware analysis on *bare-metal* (Kirat et al., 2011), i.e., on real devices rather than virtualized environments. To our knowledge, a similar approach has not been proposed or implemented for Android yet. From the hardware perspective, it would be feasible though, building upon the possibility to root devices and by communicating with an analysis PC via Android's debugging bridge.

Additionally, also for desktop PCs, Sun et al. proposed a system to detect virtualization resistant behavior in malware (Sun et al., 2011). To this end, they measure the distance in behavior between running an app on real devices and running it inside VMs. A similar approach is taken by Lindorfer et al. (Lindorfer et al., 2011). Their tool figures out evasive malware by running in different environments. However, this tool does not compare the run to a real device, but compares runs on different VMs. In the future, similar approaches could also be taken on Android to tackle split-personality malware and fingerprinting.

As opposed to the aforementioned improvements to run Android malware on bare-metal, another future research project could involve the implementation of a transparent

malware sandbox in software. As seen in Sect. 4, today it is rather simple to fingerprint available Android sandboxes. The implementation of a solution that is almost indistinguishable from real hardware remains an interesting future task. As a starting point, the fingerprinting tool Sand-Finger could be used as an opponent that must be defeated. Obvious flaws such as the missing ability to ping hosts on an unaltered QEMU emulator must be addressed. In addition, personal phone data could be faked, like sensor data, photos and contacts.

7.3. *Secure IPC mechanisms for Android*

As shown by collusion attacks in Sect. 5.1, IPC mechanisms on Android can be used to unite the permissions of two distinct apps. An app can use the permissions of another app without a user's approval. To tackle this problem, *XManDroid* (Bugiel et al., 2012) has been proposed by Bugiel et al. to patch the Android platform against the risk of privilege escalations (Davi et al., 2011). XManDroid claims to also stop collusion attacks that communicate via channels other than Android's standard IPC mechanism.

8. Conclusions

On-device AV scanners for Android have little possibilities of counteracting threats. Once a malware sample is installed, there are no possibilities to analyze it at runtime, i.e., no possibilities to detect its malicious behavior. As a consequence, the Google Bouncer was introduced to reject suspicious apps and to keep the Play Store clean. As shown by our work, however, the Google Bouncer can be surpassed by dynamic code loading attacks today. Even if dynamic code loading would be prohibited or restricted on Android in future, as proposed in the literature (Poeplau et al., 2014), split-personality malware cannot be stopped as shown by dynamic script loading attacks.

As part of this work, ten unique sandboxes have been fingerprinted using a custom-built fingerprinter called *Sand-Finger*. Moreover, we argued that results of our fingerprinter can be applied to Android malware to perform split-personality attacks that defeat automated analysis frameworks. To substantiate our approach, we demonstrated that dynamic code loading can be used to bypass the Google Bouncer, and successfully uploaded an Android root exploit to the Play Store. We also revised the effectiveness of so-called *collusion attacks* based on the AV detection rates of VirusTotal.

Furthermore, we have studied the widespread of dynamic code loading among 22,032 benign and 14,885 malicious apps in the wild. The results show that benign apps use these techniques less frequently than malicious ones, among which 36.4% were found to dynamically load and execute code by means of class loaders and the Runtime.exec() method.

To conclude, neither automated static nor dynamic analysis of Android APKs can be considered entirely secure today and foreseeable countermeasures are just the next step in an ongoing game of cat-and-mouse. Static analysis has no possibility of analyzing code or scripts which an app dynamically downloads or unpacks at runtime before execution. Dynamic analysis, on the other hand, can easily analyze code and scripts loaded at runtime, but it cannot analyze traces which are not executed inside an analysis environment. Hence, from an attackers point of view, the trick is to behave differently in analysis environments than on real devices.

Acknowledgments

The research leading to these results was supported by the *Bavarian State Ministry of Education, Science and the Arts* as part of the FORSEC research association. We want to thank Johannes Götzfried, Andreas Kurtz, Mykola Protsenko and Michael Spreitzenbarth for proofreading this article and giving us valuable hints for improving it. A special thanks also goes to Sebastian Poeplau and his co-authors for early access to their related paper (Poeplau et al., 2014).

REFERENCES

Arp D, Spreitzenbarth M, Hubner M, Gascon H, Rieck K. DREBIN: effective and explainable detection of android malware in your pocket. In: Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS), San Diego, CA; 2014.

Balzarotti D, Cova M, Karlberger C, Kruegel C, Kirda E, Vigna G. Efficient detection of split personalities in malware. In: Proceedings of the Symposium on Network and Distributed System Security (NDSS), San Diego, CA; 2010.

Bayer U, Habibi I, Balzarotti D, Kirda E, Kruegel C. A view on current malware behaviors. In: Proceedings of the 2Nd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More, LEET'09. Berkeley, CA, USA: USENIX Association; 2009. 8–8.

Becher M, Freiling FC. Towards dynamic malware analysis to increase mobile device security, in: A. Alkassar, J. H. Siekmann (Eds.), Sicherheit, Vol. 128 of LNI, GI, pp. 423–433.

Becher M, Hund R. Kernel-level interception and applications on mobile devices. Tech. rep. University of Mannheim; may 2008.

Bencsáth B, Pék G, Buttyán L, Felegyhazi M. sKyWIper (a.k.a. flame a.k.a. flamer): a complex malware for targeted attacks. 2012. Tech. rep., In collaboration with the sKyWIper Analysis Team.

Bläsing T, Batyuk L, Schmidt A-D, Camtepe S, Albayrak S. An Android application sandbox system for suspicious software detection. In: Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on; 2010. p. 55–62. http://dx.doi.org/10.1109/MALWARE.2010.5665792.

Bodden E, Sewe A, Sinschek J, Oueslati H, Mezini M. Taming reflection: aiding static analysis in the presence of reflection and custom class loaders. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11. New York, NY, USA: ACM; 2011. p. 241–50. http://dx.doi.org/10.1145/1985793.1985827.

Botnet Research Team. SandDroid: an APK analysis sandbox. Xi'an Jiaotong University; 2014. URL, http://sanddroid.xjtu.edu.cn.

Bugiel S, Davi L, Dmitrienko A, Fischer T, Sadeghi A-R, Shastry B. Towards taming privilege-escalation attacks on Android. In: 19th Annual Network & Distributed System Security Symposium, NDSS '12; 2012.

Chen X, Andersen J, Mao Z, Bailey M, Nazario J. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In: Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE

International Conference on; 2008. p. 177–86. http://dx.doi.org/10.1109/DSN.2008.4630086.
Chin E, Wagner D. Bifocals: analyzing webview vulnerabilities in android applications. In: Proceedings of the 14th International Workshop (WISA); 2013. p. 138–59.
Davi L, Dmitrienko A, Sadeghi A-R, Winandy M. Privilege escalation attacks on Android. In: Proceedings of the 13th International Conference on Information Security, ISC'10. Berlin, Heidelberg: Springer-Verlag; 2011. p. 346–60.
Anthony Desnos, Androguard: reverse engineering, malware and goodware analysis of Android applications. URL code.google.com/p/androguard/.
Enck W, Gilbert P, Chun B-G, Cox LP, Jung J, McDaniel P, et al. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI'10. Berkeley, CA, USA: USENIX Association; 2010. p. 1–6.
Fedler R, Kulicke M, Schütte J. An antivirus API for Android malware recognition. In: Malicious and Unwanted Software: "The Americas" (MALWARE), 2013 8th International Conference on; 2013. p. 77–84. http://dx.doi.org/10.1109/MALWARE.2013.6703688.
Fedler R, Kulicke M, Schuette J. Native code execution control for attack mitigation on Android. In: Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices, SPSM '13. New York, NY, USA: ACM; 2013b. p. 15–20. http://dx.doi.org/10.1145/2516760.2516765.
Fedler R, Schütte J, Kulicke M. On the effectiveness of malware protection on Android. Tech. rep. Berlin (: Fraunhofer AISEC; 2013c.
ForSafe. Forsafe mobile security, Whitepaper. 2013. URL, http://www.foresafe.com/ForeSafe_WhitePaper.pdf.
Gartner. Gartner says smartphone sales accounted for 55 percent of overall mobile phone sales in third quarter of 2013. Nov. 2013. URL, http://www.gartner.com/newsroom/id/2623415.
Hao H, Singh V, Du W. On the effectiveness of api-level access control using bytecode rewriting in android. In: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security, ASIA CCS '13. New York, NY, USA: ACM; 2013. p. 25–36. http://dx.doi.org/10.1145/2484313.2484317.
Kirat D, Vigna G, Kruegel C. Barebox: efficient malware analysis on bare-metal. In: Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC '11. New York, NY, USA: ACM; 2011. p. 403–12. http://dx.doi.org/10.1145/2076732.2076790.
Lindorfer M, Kolbitsch C, Milani Comparetti P. Detecting environment-sensitive malware. In: Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection, RAID'11. Berlin, Heidelberg: Springer-Verlag; 2011. p. 338–57.
Lindorfer M. Andrubis: a tool for analyzing unknown android applications. Jun. 2012.
Lockheimer H. Android and security. Feb. 2012. URL, googlemobile.blogspot.de/2012/02/android-and-security.htm.
Luo T, Hao H, Du W, Wang Y, Yin H. Attacks on WebView in the Android system. In: Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC '11. New York, NY, USA: ACM; 2011. p. 343–52. http://dx.doi.org/10.1145/2076732.2076781.
Maier D, Müller T, Protsenko M. Divide-and-Conquer: why android malware cannot be stopped. In: Research S, editor. 9th International Conference on Availability, Reliability and Security; 2014.
Marforio C, Ritzdorf H, Francillon A, Capkun S. Analysis of the communication between colluding applications on modern smartphones. In: Proceedings of the 28th Annual Computer Security Applications Conference, ACSAC '12. New York, NY, USA: ACM; 2012. p. 51–60.
Moser A, Kruegel C, Kirda E. Limits of static analysis for malware detection. In: Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual; 2007. p. 421–30. http://dx.doi.org/10.1109/ACSAC.2007.21.
Neugschwandtner M, Lindorfer M, Platzer C. A view to a kill: webview exploitation. In: Presented as part of the 6th USENIX Workshop on Large-Scale Exploits and Emergent Threats. Berkeley, CA: USENIX; 2013.
Oberheide J, Miller C. Dissecting the Android Bouncer. Presented at SummerCon 2012, Brooklyn, NY. 2012. URL, https://jon.oberheide.org/files/summercon12-bouncer.pdf.
Percoco NJ, Schulte S. Adventures in bouncerland: failures of automated malware detection with in mobile application markets. In: Black Hat USA '12, Trustwave SpiderLabs; 2012.
Petsas T, Voyatzis G, Athanasopoulos E, Polychronakis M, Ioannidis S. Rage against the virtual machine: hindering dynamic analysis of android malware. In: Proceedings of the Seventh European Workshop on System Security, EuroSec '14. New York, NY, USA: ACM; 2014. 5:1–5:6. http://dx.doi.org/10.1145/2592791.2592796.
Poeplau S, Fratantonio Y, Bianchi A, Kruegel C, Vigna G. Execute this! Analyzing unsafe and malicious dynamic code loading in android applications. In: Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS), San Diego, CA; 2014.
Protsenko Mykola, Müller Tilo. PANDORA applies non-deterministic obfuscation randomly to Android. In: Osorio Fernando C, editor. 8th International Conference on malicious and Unwanted software; 2013.
Raffetseder T, Kruegel C, Kirda E. Detecting system emulators. In: Proceedings of the 10th International Conference on Information Security, ISC'07. Berlin, Heidelberg: Springer-Verlag; 2007. p. 1–18.
Rastogi V, Chen Y, Jiang X. Droidchameleon: evaluating Android anti-malware against transformation attacks. In: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security, ASIA CCS '13. New York, NY, USA: ACM; 2013. p. 329–34. http://dx.doi.org/10.1145/2484313.2484355.
Shabtai A, Fledel Y, Kanonov U, Elovici Y, Dolev S, Glezer C. Google Android: a comprehensive security assessment. IEEE Secur Priv 2010;8(2):35–44. http://dx.doi.org/10.1109/MSP.2010.2.
Shabtai A, Kanonov U, Elovici Y, Glezer C, Weiss Y. "Andromaly": a behavioral malware detection framework for Android devices. J Intell Inf Syst 2012;38(1):161–90. http://dx.doi.org/10.1007/s10844-010-0148-x.
Spreitzenbarth M, Freiling FC, Echtler F, Schreck T, Hoffmann J. Mobile-sandbox: having a deeper look into Android applications. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13. New York, NY, USA: ACM; 2013. p. 1808–15. http://dx.doi.org/10.1145/2480362.2480701.
Spreitzenbarth M. Dissecting the Droid: Forensic analysis of android and its malicious applications. Ph.D. thesis. Friedrich-Alexander-Universität Erlangen-Nürnberg; 2013.
Sun M-K, Lin M-J, Chang M, Laih C-S, Lin H-T. Malware virtualization-resistant behavior detection. In: ICPADS. IEEE; 2011. p. 912–7. http://dx.doi.org/10.1109/ICPADS.2011.78.
Vallee-Rai R, Co P, Gagnon E, Hendren L, Lam P, Sundaresan V. Soot – a java bytecode optimization framework, In: Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research, CASCON '99, IBM Press.
Van der Veen V. Dynamic analysis of android malware. Master Thesis. VU University Amsterdam; Aug. 2013. URL, http://tracedroid.few.vu.nl/thesis.pdf.

Vidas T, Christin N. Evading android runtime analysis via sandbox detection. In: Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14. New York, NY, USA: ACM; 2014. p. 447–58. http://dx.doi.org/10.1145/2590296.2590325.

Willems C, Holz T, Freiling FC. Toward automated dynamic malware analysis using cwsandbox. IEEE Security and Privacy 2007;5(2):32–9. http://dx.doi.org/10.1109/MSP.2007.45.

Dominik Maier studies Computer Science at the University of Erlangen-Nuremberg. In 2014 he received his B.Sc. in Business Informatics from the University of Erlangen-Nuremberg. Concurrently, as part of a combined study program, he held an apprenticeship with DATEV eG and earned his "Computer Science Expert"-certificate. He worked in the research department in the field of computer security, with a focus on secure authentication and mobile security.

Mykola Protsenko was born in Dnipropetrovsk, Ukraine. He studied dynamics and durability at the National University of Dnipropetrovsk from 2002 to 2004 and Computer Science at the University of Erlangen-Nuremberg, where he received his Diploma with distinction in 2013. Currently Mykola Protsenko is a PhD Student at the chair for IT Security Infrastructures at the University of Erlangen-Nuremberg. His main research interests include software protection and applied IT security.

Tilo Müller studied Computer Science at the University of Aachen and the University of Helsinki. He received his Diploma from the University of Aachen in 2010 and since then, he is employed as a research assistant at the chair for IT Security Infrastructures in Erlangen. Tilo Müller received his doctoral degree in 2013. His research interests include system security, mobile security and software protection.