Common Weaknesses of Android Malware Analysis Frameworks

Lars Richter
University of Erlangen-Nuremberg

***Abstract*—In order to evade anti-malware products of different vendors, Android malware authors are seeking for possibilities to gain information about the execution environment of their applications. So called split-personality malware loads additional code during runtime to prevent detection by offline code analysis e.g. the Google Bouncer. To evade detection during runtime it behaves like a normal app and analyzes its environment at first. If an analysis environment can be excluded the app will load and execute the malicious code. To prevent such analysis by malware an Android sandbox seeks to simulate the real smart phone as close as possible and leaves minimum trace of the virtualization. In previous work different Android sandboxes where fingerprinted to detect the analysis environment. In this paper we are building on these findings to present and categorize different weaknesses of Android malware analysis frameworks. With that knowledge it is possible to improve Android sandboxes to spoof e.g. split-personality applications to execute their malicious code and thus detect them.**

I. Introduction

With its market share of 78% in shipped units Android is dominating the smartphone market in the first quarter of 2015. Without a doubt Android has become the most popular Operating System for smartphones and tablets. With the increase of computing power and rising functionality of the operating system and its applications, people rely on these devices for all Internet-related subjects. From messaging to shopping and banking, users confide their mobile computers their personal secrets and login credentials. Criminals are taking advantage of that trust with more and more sophisticated malware. In contrast to other smartphone operating systems android allows the user to install software from unverified sources. This is on the one hand a big advantage over the other systems, because the user can decide which software he wants to install without loosing the warranty of the device. As a consequence, multiple app stores for Android devices exist. On the other hand enables such an distributed ecosystem multiple starting points for spreading malicious software *AndRadar*. Each store has to deal with malicious applications and must detect them to preserve their reputation.

A. Motivation

According to recent studies over 2000 new malicious applications are discovered everyday[36]. It is obvious that there is no possibility to detect such a number of new malware manually. Google developed the *Bouncer* as a first action to detect malware. The Bouncer is a service that analysis every submitted application to the Google Play Store. If an application is seen as malicious it is rejected and the developers account, and all accounts issued from that IP, will be automatically banned [24]. According to Google the introduction of the Bouncer resulted in a 40% reduce of malware in the store [14]. Alternative markets which do not have such anti malware measures in place contain 5-8% malicious apps [18]. To analyze newly submitted applications and to detect malware, multiple automated frameworks were developed by researches and security companies [1][16][17][28][34][40]. These anti malware frameworks are using different approaches and detection mechanism. In a recent study, multiple malware analysis frameworks were compared according to their analysis approach, technical / logical scope, delicateness to known bugs, behavior representation and detection [23]. Each analysis approach has its advantages and disadvantages. But there is also malware which can not be detected with known frameworks [19]. Highly sophisticated malware can simulate benign behavior and executes its malicious hidden content, if it is installed on an non analysis environment. Because of the heterogeneous and diverse analysis frameworks the reasons malware remain undetected are numerous.

B. Contribution

In this paper we are investigating the reasons some malware stays undetected in some analysis environments. We characterize the different analysis types, we present some popular frameworks which are based on these types and we describe some drawbacks of each analysis type. One disadvantage of the static analysis is e.g. the inability to scan highly obfuscated code. Transformations form the basis of the obfuscation and they are described in section III. In the following section IV we demonstrate the numerous possibilities an Android application can fingerprint its host. It becomes clear that it is a serious task to trick an application into believing it is executed on an physical host during the dynamic analysis to observe its malicious behavior. Most analysis approaches assume that an applications is fully sandboxed and can not communicate with other applications to gain higher privileges. In section V we give an overview about common communication channels caused by application collusion and the risks. A further difficult malware behavior is described in section VI. Unforeseeable Events, like external events or timing events can not be simulated due to their complexity or resource

This paper was written as part of the conference seminar "IT Security" which was organized by the Chair for IT Security Infrastructures (Prof. Dr. F. Freiling) at the University of Erlangen-Nuremberg during summer term 2015

limitations. Malware can use these events to trigger malicious behavior. We present an introduction to *HARVESTER*, which claims to be able to defeat these mechanism. As an prospect to related and for future work we summarize in section VII improvements of the different analysis methods to overcome the listed weaknesses. We characterize the approach of the *BareCloud* as one solution to benefit from the highly fragmented malware analysis environment. The investigations scope is limited to the referenced literature.

C. Related Work

Poeplau et al. [26] developed a static analysis tool for detecting dynamic code loading of Android applications during runtime. They performed a study with that analysis and they found out, that 8 of the top 50 free Android applications are using dynamic code loading. They also showed that code loading can exploit a conceptional weakness of analysis frameworks.

Another research field are the possible communication channels which are used by colluding applications on an Android smartphone. Schlegel et al. [32] presented *Soundcomber* as a first Trojan which uses covert communication channels to exchange information. Therefore the Trojan itself is only using a small amount of permissions. Another application is waiting for the data on the covert channel to send it over the network. Both applications in particular are unsuspicious, but when combined, they pose a serious threat. Marforio et al. [20] analyzed the possibilities to use communication channels to transfer data between colluding applications. Analysis tools failed to detect the data exchange. Therefore they conclude that covert communication channels are a threat to smartphone security. Mazurczyk and Caviglione [21] complemented the work of Marforio et al. They survey the different methods and approaches to hide information on the smartphone. They also reviewed the methods according to detection possibilities.

Rastogi, Chen and Jiang developed the *DroidChameleon* [30], a systematic framework to evade detection by static analysis frameworks. They describe different transformation attacks to reduce the signatures commercial anti malware tools can detect. They state that simple transformations are successful because most analysis tools are searching for known signatures and are prone transformation attacks.

Another possibility to evade detection by analysis systems is to fingerprint the analysis environment first and hide the malicious intent of the application. Petsas et al. [25] presented detection heuristics of dynamic analysis environments of three categories, static properties, dynamic sensor information and VM-related complications. With the help of these heuristics they were able to avoid detection of analysis systems. Vidas and Christin [31] [39] also showed different techniques for detecting Android analysis systems. They classified their approaches into behavior, performance, hardware- / software-components and analysis system design choices. They evaluated the detecting techniques against the analysis frameworks *Andrubis, CopperDroid* and *ForeSafe*. Maier, Müller and Protsenko [19] demonstrated how malware can evade analysis systems and presented a tool for fingerprinting multiple Android-based analysis systems. They where able to create a malware that successfully surpasses existing malware scanners and they successfully bypassed the Google Bouncer by uploading a modified Android root exploit. Balzotti et al. [3] presented a first approach for Android which is able to detect split-personality malware, by executing the malware in an emulator and on an uninstrumented reference system.

Malware analysis systems can by divided into static and dynamic analysis systems. *Drebin* [1] performs a fast broad static analysis on the phone during runtime, and searches for patterns of malicious applications with the help of machine learning. Like *Drebin* is *Marvin* [16] another on-device analysis tool. It is using similar approach to classify applications based on a set of features which are extracted during a static and dynamic analysis. For the dynamic analysis the application file in question is submitted via a web interface or the Marvin application itself.

The dynamic analyzes executes the malware in a monitored and often sandboxed environment. Hybrid approaches are using static analysis to improve the dynamic execution. *Andrubis* [17] is a fully automated hybrid analysis systems. Andrubis is publicly available and was able to collect analysis data of over 1 million applications, where 40% had a malicious intend. That dataset is used to discuss trends in applications behavior to differentiate between benign and malicious. *Andlantis* [4] is a highly scalable dynamic analysis frameworks which is able to schedule and analyze thousands of Android instances in parallel to make best use of the limited computational resources. To detect privilege escalation attacks through covert channels *XManDroid* [5] extends the monitoring mechanisms of Android. The implementation dynamically analyzes the permission usage of the different applications and communication links between the applications. *TaintDroid* [8] is an information tracking analysis system which monitors multiple sources of sensitive data. With *TaintDroid* it is possible to detect possible misuse of sensitive data by third party applications. Another taint analysis framework is *FlowDroid* [35] which monitors callbacks by the Android framework and uses context, flow, field and object-sensitivity to reduce the number of false positives. The *Mobile-Sandbox* by Michael Spreitzenbarth et al. [34] is also a hybrid analysis approach. The static analysis is used to reach higher code coverage during the dynamic analysis. Additionally it uses specific techniques to log native API calls, which can be used to hide malicious content. They found that the existence of native code calls does not imply that the application is malicious.

The *BareCloud* [15] is an automated evasive malware detection system which is using multiple analysis approaches, including a bare-metal reference system. It observes the malware behavior on the different systems and compares them to detect split-personality malware. This approach is focused on Windows malware. Balzotti et al. [3] presented a similar approach to detect Android malware. Rasthofer et al. developed the *HARVESTER* [28], which is an approach to defeat split personality malware. It uses program slicing and dynamic execution to extract runtime values from any position in the Java bytecode. This analysis approach is very effective against highly obfuscated malware which uses timing and logic bombs.

Other researchers analyzed and compared multiple frameworks with each other. That research is used to get an overview about the similarities and differences of the analysis approaches. Fedler, Schütte and Kulicke [10] evaluated multiple antivirus applications in regards to their analysis approach and detection rate. They conclude that antivirus software may be reliable to detect long-known threats but the capability to detect new threats or variants of existing malware is limited. Neuner et al. [23] compared 10 dynamic Android analysis sandboxes in terms of feature support and the analyzed application properties. They evaluated their effectiveness with known malware samples and Android bugs. Lindorfer et al. presented with *AndRadar* a framework for discovering malicious applications in different Android markets to expose the distribution strategies of malware authors. They evaluated how fast markets detect and delete malware.

II. Categorization

As previously motivated there is a need for analyzing smartphone applications. In general the analysis can be differentiated into *static* and *dynamic* analysis. The use of both techniques is called *hybrid* analysis.

A. Static Analysis

The static analysis covers aspects of the application without actually executing them. A key artifact which is analyzed by many frameworks is the manifest file which is required by the application. The *AndroidManifest*[1] provides meta information about the unique package name, used activities, services, broadcast receivers and content providers. It names classes which implement these components and publishes their capabilities. With that information the Android system knows under which condition each component has to be launched. Additionally the manifest defines which permissions are needed to access protected parts of the API. The access to specified hardware components can be an indicator of malicious behavior. A well known example is the torch app[33] which requests GPS and network access to send the users location data to the attacker. Another hint for malware is the *SEND_SMS* permission which is often used to send premium SMS.

Another approach is to analyze the byte code of the applications. Since the code is not executed, and no variables are set, it can not be decided which paths will be taken by the application. With the help of graphs analysts can understand the inner working of an application and how the code blocks are connected [13]. Suspicious API calls which access sensitive information can be detected with that approach. API calls which encrypt or decrypt data or execute external code are often used for code obfuscation but can also be detected with the static analysis [1]. Obfuscation will be discussed in detail in section III. External code can be found by checking each resources file type in the Android Application Package (APK). Malware hides often libraries in seemingly benign external files to avoid detection of suspicious API calls. Android programs are compiled into *Dalvik Executable Files* (.dex Files). The dissembled .dex files can be searched for strings. These strings can be scanned for IP addresses, which could point to command & control servers or data sinks for private information. A well known tool for static code analysis is *Androguard* which disassembles and decompiles Dalvik byte code to Java Source Code. Frameworks like *Tracedroid*, *Andrubis* and *Sanddroid* are using that static code analyzer [23].

[1] https://developer.android.com/guide/topics/manifest/manifest-intro.html

B. Dynamic Analysis

The dynamic analysis approach involves the execution of the application on either a virtual machine or a physical device. During the analysis, the behavior of the application is observed and can be analyzed. The dynamic analysis results in a less abstract view of the application than the static analysis. The code paths executed during runtime are a subset of all available paths. The main goal for analysis frameworks is to reach high code coverage because all possible actions should be triggered to observe any possible malicious behavior. Research has shown that fully randomized input achieves a 40% or lower code coverage. [12] *Multipath execution* is a way to increase the code coverage. Whenever a branch is taken, the current state of the VM is saved in a snapshot so that it can be rolled back and execute the other branch. However this is only partially applicable because this behavior much likely breaks network protocols. [22] Depending on the data of interest, different techniques exist to monitor the applications behavior. One analysis technique is taint tracking. A system wide implemented taint propagation is able to analyze the message flow and potential misuse of private sensitive information through third-party applications [23]. A popular framework which uses that technique is *TaintDroid*. Developed with the Dalvik Virtual Machine, it monitors how applications access and manipulate user data in real time. It labels the sensitive data as it flows through variables, files and messages. However TaintDroid is only able to detect explicit data flow and is not able to analyze implicit flow through control flow. Private information could be transmitted over that channel. [8]

Another analysis technique is *virtual machine introspection* (VMI) which is used to intercept events within the emulated environment. It is also used to monitor the execution of the Android API. VMI is either possible by modifying the Dalvik VM or by using the QEMU emulator itself. A further possibility to collect executed system calls is the standard Linux library trace tool *ltrace* used by the Mobile Sandbox [34].

C. Discussion

Static code analysis can be used to get an overview of the applications but is very abstract. It can be used to reveal the leakage of sensitive information, inter process communication, network communication and cryptography misuse and more [9]. However the static analysis can be tricked by obfuscation techniques. Dynamic approaches on the other side can deliver precise results because runtime data and values are available and can be retrieved. But that data depends highly on the taken

code paths in contrast to the results from the static analysis, which are produced by the whole codebase. Therefore the best approach is the hybrid approach which uses static code analysis to gather information to improve the outcome of the dynamic analysis. *Andrubis* for example compares the requested permissions in the manifest with the permissions that are requested in the byte code, as well as the used ones during the execution in the dynamic analysis [17].

III. Code Obfuscation

The goal of code obfuscation is to prevent static code analysis. Code obfuscation alone can not prevent detection of malicious behavior through dynamic analysis because the code will be deobfuscated during execution and therefore the function calls and values can be monitored. We will describe a small number of transformation attacks which sufficiently hinder the analysis by static analysis frameworks.

A. Trivial Transformations

Trivial transformations are mainly attacking signature based detection approaches. Android packages are signed zip files. These packages can be unzipped, repacked again and signed with a custom key. In addition the package name can be altered. The detection approaches which are based on the package signature or a hash of the complete app will fail. Signatures on the bytecode can be defeated with disassembling, reordering and reassembling again. Signatures based on single items will be unusable. [31]

B. Detectable Transformations

Some transformation attacks can be revealed by the data flow, which is not altered by these attacks. These transformations are detectable by Static Analysis (DSA) [31]. Methods for DSA transformation attacks are renaming of static strings (such as classnames and methodnames), code reordering, changing of the call directions, junk code insertion, data encoding and encrypting payloads and native code exploits. The last method describes the hiding of native code exploits in non standard locations in the applications package. These exploits are stored encrypted an will be decrypted during runtime. This method differs from the *Bytecode Encryption attack*, which is a non-detectable transformation attack because the main application can still be analyzed. [31]

C. Non-Detectable Transformations

These methods are preventing the static code analysis. The analysis could detect that the following methods are executed but it can not decide whether its cause is benign or malicious. It would result in a high false positive rate.

The Java Reflection API allows to modify the runtime behavior of applications.[2] The method call could be changed into any other call during execution. This defeats the data flow analysis of static frameworks.

[2]https://docs.oracle.com/javase/tutorial/reflect/index.html

Another transformation is *Bytecode Encryption*. The main function of the application is stored in a separate dex file in encrypted form. The only parseable code is the decryption routine. With the help of the *DexClassLoader* the external dex file can be loaded and executed. Analysis frameworks could detect the usage of the DexClassLoader but only a dynamic analysis system can investigate the behavior further.

The DexClassLoader can be used to execute downloaded code too. Benign applications use that procedure to install add-ons so it cannot be flagged as malicious by default. In addition applications can directly request the installation of a downloaded APK which prompts the user a dialog. Poeplau et al. exploited that method to build an application which downloads and executes malicious code [26]. Their app was not detected as malicious by the Google Bouncer or any anti virus application.

D. Discussion

The transformation attacks are a significant threat to static analysis frameworks. Simple package and identifier renaming techniques allow evasion of some analysis tools. More sophisticated methods can be used to thwart the analysis. [31]

An example for applied code obfuscation techniques is the *Android/BadAccents* malware. The malware is using email for sending sensitive user data. The malware analyst is interested in the specific API calls for sending the email. Therefore he searches with the help of static code analysis for these API calls. But the malware is saving the sensitive information as native code. Static and forward analysis can not be used to extract the values from ARM native code, therefore the API call information will remain undetected. [29]

Poeplau et al. described code injection against benign applications [26]. The application developers are in duty to check if their downloaded code is integer and authentic. The technical understanding for the security risks is often not existent or there is no business value in securing their applications. In consequence an attacker is able to replace downloaded code with its own malicious code. This could happen during insecure HTTP downloads, on the unprotected storage on the smartphone or the improper use of package names. With that code injection technique the attacker can use the allowed permissions of the benign app to gather sensitive information to his own benefit. The risks of external code execution are often underestimated. The attack vector of exploiting known benign applications or underlying libraries by injecting malicious code is not observed by current analysis frameworks.

IV. Fingerprinting

As already mentioned static analysis approaches can be made useless with code obfuscation. Code obfuscation can be defeated with the dynamic analysis approach. Highly sophisticated malware could be able to detect the monitoring and thus take actions to prevent the analysis of its malicious behavior. This method is called *Fingerprinting* and will be described in the following.

A. Problem Statement

In contrast to desktop PCs the operating system of a smartphone is aware of the build-in hardware because it is not replaceable. There is no need to support modular hardware. In consequence many different kernels to support the different devices, specialized to each hardware configuration, are existing. If an app is installed on a normal smartphone, it expects that it can use the camera, GSM modem, GPS and other sensors.

In a desktop PC environment analysis frameworks could simply disable the network card because there are computers which are not connected to any network, and malware authors could target these PCs too. But it is very uncommon that a smartphone got a disabled GPS sensor or camera. Even more suspicious is GPS data which is fully randomized or camera pictures which are always the same. Analysis environments should be indistinguishable from the real device.

The problem is, that in contrast to PC sandboxes, all sensors and hardware components must behave like their counterparts on the physical devices. There is no modularity where a component can be disabled. E.g. a smartphone without a SIM card would raise a red flag to malware because it is most likely executed in a sandbox.

This task gets more difficult because of the numerous device information, settings and saved connections. A malware author can define numerous conditions which certainly indicate the environment as physical device. This leads to the problem that the malware behaves benign in an analysis environment and malicious at physical devices [3]. Maier, Müller and Protsenko presented Sand-Finger a tool that finds characteristics where analysis environments differ from the real device [19]. They cannot find any sandbox which is indistinguishable from real hardware.

B. Possibilities

There are numerous possibilities to differentiate between the real device and the virtualization environment. They can be divided into several groups.

Sensor behavior - describes the values which are returned by the sensors of a smartphone, e.g. GPS sensor, acceleration sensor, brightness sensor, temperature sensor and barometer. These values must be consistent and realistic. Temperature changes in milliseconds are as unrealistic as a z-acceleration by 0, which means the device is situated in zero gravity.

User interaction - is the way the sensors work together and the user navigates through the interfaces. Many frameworks use the Monkey Runner[3] to simulate semi random user input from a given seed. That approach is sometimes not practicable. The malware Android/BadAccents requires the user to put in his bank account password twice and some numbers which have to match a defined pattern [29]. If these conditions are not met no malicious actions will be performed. The behaviour of the monkey runner could also be exploited. Andrubis [17] and Tracedroid [37] are using the same seed for every execution. The malware author can execute the monkey runner with the same seed and save this pattern in the app. He can following flag that behavior as suspicious and hold back any malicious behavior and therefore prevent any detection.

Hardware Components - Since the application has access to I/O ports, memory management chips and networking devices, these all must be made available by the emulation environment. CPU frequencies are available on the physical device but are not available by default in the emulator [39]. The responses to low level calls must behave like the function calls on the real devices.

Android API - many API methods return fixed values in a VM. For example is the return value of *Build.HARDWARE* ”goldfish” in the standard Android Emulator. The combination of some values clearly indicates that an emulator is used. [39]

Protocol implementations - differ from real devices too. The Android Emulator cannot forward pings. In a recent study only one out of 10 sandboxes are able to emulate a ping to a destination [19] and is therefore an easy way to spot the emulation.

Virtualization overhead - exists e.g. in QEMU because of the binary translation. QEMU translates the code block wise and executes it. During the execution of the block is no possibility to handle an interrupt by the host OS. Therefore it is easily detectable [11]. A solution is QEMU’s single step mode, which defeats this detection but is noticeable slower and again detectable[39].

Personal Information - Malicious applications could expect some personal information on the devices, like contacts, messages, missed calls, pictures, saved WLAN networks, paired bluetooth devices, music files or a browser history. If some data did not exist or the samples are too old, it is much likely an unused device and therefore not worth the risk to be discovered by an analysis framework.

Another simple indicator for a sandbox is the uptime. During the analysis the emulated device is often rebooted to create a replicable environment. Normal devices are running the whole day, so an uptime is much likely higher than 10 minutes. The most sandboxes do not manipulate the uptime [19].

Maier, Müller, Protsenko conclude that the combination of uptime, a hardware string and a list of connected networks is a reliable indicator for the execution environment [19].

Petsas et al. [25] classify the described detection types into 3 categories. Static heuristics are based on static information which are fixed values in the emulation environment. Dynamic heuristics are based on unrealistic behavior of the sensors. Hypervisor heuristics describe incomplete emulations of the hardware. They evaluated multiple sandboxes for their sturdiness against multiple evasion techniques.

C. Discussion

As previously stated simple measures can be taken to detect an analysis environment. Malware authors and sandbox developers are in an arms race to defeat each other detection mechanisms. PC malware is not using fingerprinting methods so intensively because PC virtualization is a standard technique in data centers. So the malware has to be executed despite its

[3]https://developer.android.com/tools/help/monkeyrunner_concepts.html

environment because VMs are rewarding hosts. There is no practical use-case for smartphone virtualization to this time. So a malware author has no intention in infecting a known virtualized host because of the risk of detection [39].

Another way to prevent fingerprinting is to use an analysis system which is indistinguishable from the physical host. Such systems are known as transparent analysis systems. *Cobra* [38] performs dynamic translation of the code during runtime. Instructions which could be used to detect the virtualization are replaced with save ones. It could only replace known fingerprinting methods. *Ether* [7] uses a more transparent approach by using the hardware virtualization of the CPU. That virtualization comes with the cost of performance which could be detected by timing analysis.

However malware could implement checks for anomalies in the internal or external environment that detects analysis frameworks. It could wait a specific time for real user activity or trying to connect to a non existent domain. If there is no real user interaction like long device sleeps during the night and every non-existent domain is resolved, the analysis environment is detected.

To avoid the fingerprinting Kirat and Vigna presented the *BareCloud* [15], a bare-metal based malware detection based on the *Cuckoo Sandbox*[4] for Windows. The use of physical devices instead of virtualization makes the analysis more transparent and robust against highly specialized malware. The approach is to execute the malware in different environments and compare the behavioral profiles to find differences. The assumption is that the malware successfully fingerprints one sandbox and behaves benign to evade it. To detect differences the malware has to show its malicious behavior in one of the analysis systems. This system is called the reference system and is a bare-metal system to simulate the real device as close as possible. The BareCloud is to our knowledge the most successful framework for detecting high sophisticated malware. It combines bare-metal, transparent (Ether [7]), hybrid (Anubis[5]) and simple emulation frameworks (Virtual Box)[6] to detect as many differences as possible in the malware behaviour to detect sandbox evasion. Known and unknown fingerprinting methods will fail to detect the BareCloud framework because the reference device is a bare-metal analysis environment. However Kirat, Vigna and Kruegel indicate that there is a possibility to fingerprint the bare-metal device by MAC address and the presence of the iSCSI drivers. [15]

V. Application Collusion

One security mechanism on Android is the permission-based approach. An user can review the used permission by an application and can decide if the requested permissions are justified. The permission-based security approach suffers a big disadvantage when it comes to application collusion. Users are believing that they approve the requested permissions to each application independently. Researchers have found out that with the use of covert communication channels quite the contrary is the situation [20] [21].

[4]http://www.cuckoosandbox.org
[5]http://anubis.iseclab.org
[6]http://www.virtualbox.org

A. Soundcomber

The first malware using application collusion is the *Soundcomber* which records sound and sends sensitive information with the help of a covert channel to another application. Afterwards the information is sent to a data sink within the internet [32]. In consequence the Soundcomber itself only needs a few and unobtrusive permissions. The application collusion attacks are neither a software vulnerability nor related to a particular implementation. The reason for the vulnerability itself is the assumption that the applications are independently accessing the available resources and are not able to communicate with each other. Furthermore, malware analysis frameworks are using the same assumption. Therefore application collusion can not be detected when only one application at a time is analyzed.

B. Overt Communication Channels

Marforio et al. classify the communication channels based on their implementation into application, OS, hardware and based the detection possibilities into overt and covert channels [20]. An example for an overt application communication channel is a shared configuration. Two applications can asynchronously exchange information by using the Android API to store and read data at an Android preference XML file[7]. However the creation and querying of the preferences could be detected when one of the two communicating applications is installed. Another overt communication channel which can operate without the use of special permissions are Broadcast Intents. The source application communicates via a payload added to broadcast messages within the system. The sink application registers itself as receiver of these particular broadcast messages. This communication assumes that both applications are running synchronously. Again, this approach can be detected by dynamic analysis frameworks because the underlying system calls can be traced to malicious behavior.

C. Covert Communication Channels

Covert communications channels are far more sophisticated. Schlegel et al. describe and evaluate multiple different covert channels [32]. In contrast to some overt communication channels, covert communication channels are synchronous and can not save data persistently. This means the communication partners have to synchronize each other before actually exchanging information. One covert channel is the vibration setting of the device. The data source toggle the silent mode and the data sink interprets this binary information. The use of the volume settings enables a higher bandwidth because different values are possible. The achieved bandwidth rises to 150 bps from 87 bps. A higher bandwidth of 685 bps is reached by using file locks. Simplified, the sender locks a file and the sink also tries to lock it. A binary ”one” is send when the lock attempt fails and a binary ”zero” is being send the sink gets the lock. Marforio et al. [20] extend that covert methods. They achieved a throughput of 4324.13 bps on a

[7]https://developer.android.com/reference/android/content/SharedPreferences.html

Samsung Galaxy S using the type of intents[8] to send data. This channel is using the tremendous number of possibilities. An intent can be configured with actions, flags and extra data to exchange the data. This channel is similar to the overt channel where the data is stored in the payload of the intent. As described the covert communication has to be synchronized at first. Sometimes it is crucial for malware to communicate in milliseconds with a command and control server. Therefore the synchronization time of the covert channel should be taken into consideration. A high amount of data can be exchanged through *UNIX Socket Discovery* [20]. This approach is based on one synchronization and one communication socket. The source will open the synchronization socket if the communication socket can be checked. The sink interprets the status of the communication socket when the synchronization socket is open. The synchronization time of this channel is around 5ms on a Nexus One device and is able to transfer 2610bps.

D. Detection

A monitoring framework to detect covert communication channels is *TaintDroid* [8]. As mentioned in Section II-B, Taintdroid is able to track information-flow to reveal suspicious actions. However, with its taint-tracking only variables, files and interprocess messages can be monitored. Sensitive information can be leaked through control flow. A framework that extends the available monitoring mechanisms of android is called *XManDroid* (eXtended Monitoring on Android) [5]. It is able to analyze the permission usage of applications during runtime to detect application-level privilege escalation attacks. The idea is to maintain a system state which contains all executed applications and the communication links between them. XManDroid can approve or block these communication links based on permanent conditions like: "An application that is notified about incoming or outgoing calls and can record audio must not communicate to an application with network access." [5] The main task is to define rules, which are not too strict to hinder benign applications but are strict enough to prevent privilege escalation attacks through covert channels.

Mazurczyk and Caviglione [21] investigated further communication channels, or steganography methods, which are available on smartphones. They summarized their findings that smartphones will become the most targeted devices for data exfiltration because of their importance, omnipresents and sensitive sensors. Covert channels are therefore an ideal way to transport data. The lack of an available detection mechanisms for covert channel exploits results in evasion of the most malware analysis frameworks.

VI. Unforeseeable Events

Another challenge for dynamic analysis are events that are so complex that they cannot easily be triggered or one might not be able to trigger them because of the limited analysis environment resources. A dynamic analysis must execute all available paths to be complete. Measures like code obfuscation or fingerprinting can be used to minimize the code coverage of the analysis.

[8]https://developer.android.com/reference/android/content/Intent.html

A. External Events

External events are triggering *Intents* on Android. Intents are internal events which can be used to request an action from another app component. With the help of intent-filters[9] an application can define which types of intents should be delivered to the app component. The challenge for malware analysis systems is to generate external events that trigger these actions. An example intent is the receipt of a SMS. All registered intent filters will be triggered. Therefore a malicious application can behave benign until a certain intent is triggered. With help of static analysis a hybrid analysis environment can detect that the application is listening on some intents, because they must be set in the applications manifest. But it has no clue which special message will trigger a special behavior. Advanced analysis frameworks like *MARVIN* [16] or *BareCloud* [15] will flag unused intents as suspicious.

To maximize code coverage by increasing the number of intents fuzzy testing frameworks have been developed [27]. The *Intent Fuzzer* is using static analysis of the manifest and the Dalvik bytecode to build a control flow graph with the help of *FlowDroid* [35]. In the following it creates well formed intents that will trigger these actions during a dynamic analysis. The last step describes the generation of intents with randomized values. With the help of random generated intents they are able to trigger some additional behavior. However the analysis did not scale for real world applications because all explored paths must be kept in memory - which causes that FlowDroid runs out of memory.

A highly sophisticated approach can use covert channels to trigger the sleeping malware. That approach is almost impossible to detect.

B. Timing Events

Another challenge for code analysis systems are timing events. Because of the limited resources and the high number of applications, each application is only tested for a short period of time. Malware which behaves benign until a specific date will probably not be detected if the malicious code is sufficiently obfuscated or is loaded during runtime.

On February 3rd 2015 researches at AVAST reported that applications which were available on the Google Play Store, downloaded to 5-10 million devices, turned out to be malware after 30 days of installation [2].

There exists a chance of detection with the help of hybrid analysis approaches. The *Mobile Sandbox* [34] saves in the static analysis all implemented timers and intents the application uses. During the dynamic analysis the detected intents will be triggered and the analysis will be executed a certain time to trigger the implemented timers. However, if the timer is connected to other typical user behavior like long device sleeps during the night etc. it will not trigger the malicious behavior.

Rasthofer et al. present *HARVESTER*, which allows fully automatic extraction of runtime values from any position in

[9]https://developer.android.com/guide/topics/manifest/intent-filter-element.html

the Android bytecode [28]. It is a new approach to defeat highly obfuscated code and anti-analysis techniques (e.g. fingerprinting, delayed execution, Java reflection). HARVESTER combines static-analyzing by program slicing with dynamic code execution. The slicing isolates the program code which is involved in computing a specific value of interest. Other values that do not contribute to the value are dismissed. HARVESTER simulates the reaction to environment values the application implements, instead of simulating the environment values itself. Therefore HARVESTER is able to trigger different behaviors by creating parametric slices. With these precomputed slices it is possible to create a reduced APK, which contains the code that is involved in the computation of the values of interest. Secondly the dynamic analysis executes the reduced APK in an emulator or on a stock Android phone. All different behaviors of the parametric slides will be triggered to gain a complete reconstruction of the values of interest. This approach makes the need of UI interactions unnecessary and therefore increases the code coverage. With these mechanisms it is possible to improve existing frameworks like TaintDroid or FlowDroid, that cannot analyze highly obfuscated code by the Java reflection API. An analyst can use HARVESTER to produce an APK that only executes the slice which leaks sensitive data. Runtime values and reflective calls are now statically embedded. This allows the tools to discover the data flow.

VII. Improvements

As previously described the detection of malware highly depends on the level of obfuscated code and the fingerprinting possibilities of the malware. Code obfuscation can be defeated with the use of semantic-based analysis and dynamic code analysis.[6] There are also some methods like time bombs, which are only executed after a certain time has passed, or logic bombs, that are activated by external triggers. These methods still challenge current analysis frameworks.

With the help of semantic-based malware analysis the program can be seen as a network of abstract instructions. This approach is syntactic and ignores the semantic of instructions. Therefore obfuscation methods which target pattern matching analysis are impractical. Christodorescu et al. [6] are using templates to describe a definition of a variable and all its uses in the program. Because of the usage of an abstract data flow instead of static strings the analysis is not vulnerable to the previously described trivial and detectable transformation attacks [31]. Nevertheless it is difficult to detect obfuscation techniques which are based on memory reordering or changing functions to its equivalent program instructions (e.g. replace multiplication with arithmetic left shift).

The analyzing and slicing approach of HARVESTER is a new and promising approach to defeat obfuscating techniques and external event triggering. The reduced APKs which are generated by HARVESTER can by analyzed by static analysis tools. The detection rate of FlowDroid has been increased by 300% with that approach for malware samples of the *Fakeinstaller.AH* malware family. The slicing mechanism of HARVESTER improves dynamic analysis frameworks too. With its help it is possible to execute interesting code parts directly. The analysis environment does not have to wait for a certain time (to defeat time bombs), simulate real user interactions (to trigger a specific method) or set external events (to defeat logic bombs) to observe malicious behavior. TaintDroid was able to detect the data leak instantly, without the need to trigger those conditions explicitly [28]. These results have shown that HARVESTER can be and should be used to improve the detection rate of static an dynamic analysis systems.

Split-personality malware could be exposed by using analysis frameworks like the BareCloud [15]. Although the BareCloud is not available for Android yet, its analyzing approach is substantially different from the other analysis frameworks. Instead of simulating one system as close as possible to the real device and trigger as many functions as possible, the BareCloud is taking advantage of the fingerprinting and obfuscating possibilities of the malware. By using hierarchical similarity-based behavioral profile comparison, it is possible to detect differences in the execution on bare-metal devices, virtualized, emulated and hypervisor-based analysis environments. On Android the different analysis frameworks can be used to detect changes in the applications behavior. Balzarotti et al. [3] presented a first technique which uses the BareCloud approach. They identified applications that detect the presence of a sandbox (in their study the emulator-based Anubis is used) and behave differently from the execution on a reference system. Hence they are able to identify split-personality malware. Future work has to evaluate whether it is possible to build such a system to analyze Android Applications fully automated or if it generates too many false positives.

VIII. Conclusion

In this paper we have presented a broad overview about the existing Android malware analysis frameworks and their challenges.

First we gave an insight in the categorization of the different analysis approaches. We compared static analyzes to dynamic analyzes, referenced to existing analysis frameworks which implement these approaches and discussed the pros and cons.

Second we described code obfuscation as a possibility to hide malicious code from analysis. We gave an overview about the different types of transformation attacks and discussed their threat to Android security. Additionally we mentioned the use of code injection to exploit benign applications.

Third we stated the problem that it is considerable harder to defeat fingerprinting on smartphones than it is on PCs because of the numerous sensors and hardware modules which must be implemented. We described the possibilities to detect an analysis environment based on different values. With the BareCloud we proposed a possibility to use fingerprinting of malicious applications against them because they would behave differently on an emulation environment and on a physical device.

Fourth we showed different communication channels which are caused by application collusion and lead to privilege escalation. Most of the communication channels are not checked

by the analysis environments. Often only one application at a time is analyzed and therefore no information leakage to the network will be seen.

Fifth we described two other challenges for analysis environments which can not easily be triggered. An analysis environment is not able to trigger all external events by using random intents. These triggers are called logic bombs. Time bombs are actions which are performed after a certain time, which is also difficult to achieve because of limited resources. We mentioned the HARVESTER approach that allows to trigger these actions automatically.

Sixth we gave an overview about possible improvements of the current analysis frameworks. We stated that the HARVESTER looks very promising because it defeats some obfuscation and fingerprinting techniques. In combination with an analysis framework like the BareCloud it should be possible to detect more malicious applications.

REFERENCES

[1] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2014.

[2] Avast. Apps on google play pose as games and infect millions of users with adware. https://blog.avast.com/2015/02/03/apps-on-google-play-pose-as-games-and-infect-millions-of-users-with-adware/, 2015. Accessed: 2015-07-14.

[3] Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Efficient detection of split personalities in malware. In *NDSS*, 2010.

[4] Michael Bierma, Eric Gustafson, Jeremy Erickson, David Fritz, and Yung Ryn Choe. Andlantis: large-scale android dynamic analysis. *arXiv preprint arXiv:1410.7751*, 2014.

[5] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. *Technische Universität Darmstadt, Technical Report TR-2011-04*, 2011.

[6] Mihai Christodorescu, Somesh Jha, Sanjit Seshia, Dawn Song, Randal E Bryant, et al. Semantics-aware malware detection. In *Security and Privacy, 2005 IEEE Symposium on*, pages 32–46. IEEE, 2005.

[7] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62. ACM, 2008.

[8] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[9] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *USENIX security symposium*, volume 2, page 2, 2011.

[10] Rafael Fedler, Julian Schütte, and Marcel Kulicke. On the effectiveness of malware protection on android, an evaluation of android antivirus apps. *Applied and Integrated Security*, 2013.

[11] Patrick Schulz Felix Matenaar. Detecting android sandboxes. http://www.dexlabs.org/blog/btdetect, 2012. Accessed: 2015-07-14.

[12] Peter Gilbert, Byung-Gon Chun, L Cox, and Jaeyeon Jung. Automating privacy testing of smartphone applications. Technical report, Technical Report CS-2011-02, Duke University, 2011.

[13] Johannes Hoffmann. *From Mobile to Security*. PhD thesis, Ruhr-Universitt Bochum, 2014.

[14] Dan Kaplan. Google using custom malware scanner for android apps. http://www.itnews.com.au/News/289242,google-employs-bouncer-to-cleanse-android-malware.aspx, 2012. Accessed: 2015-07-14.

[15] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barecloud: bare-metal analysis-based evasive malware detection. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[16] Martina Lindorfer, Matthias Neugschwandtner, and Christian Platzer. Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. 2014.

[17] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. Andrubis-1,000,000 apps later: A view on current android malware behaviors. In *Proceedings of the the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

[18] Martina Lindorfer, Stamatis Volanis, Alessandro Sisto, Matthias Neugschwandtner, Elias Athanasopoulos, Federico Maggi, Christian Platzer, Stefano Zanero, and Sotiris Ioannidis. Andradar: fast discovery of android applications in alternative markets. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 51–71. Springer, 2014.

[19] Dominik Maier, Tilo Müller, and Mykola Protsenko. Divide-and-conquer: Why android malware cannot be stopped. In *Proceedings of the 2014 Ninth International Conference on Availability, Reliability and Security*, ARES '14, pages 30–39, Washington, DC, USA, 2014. IEEE Computer Society.

[20] Claudio Marforio, Hubert Ritzdorf, Aurélien Francillon, and Srdjan Capkun. Analysis of the communication between colluding applications on modern smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 51–60. ACM, 2012.

[21] Wojciech Mazurczyk and Luca Caviglione. Steganography in modern smartphones and mitigation techniques. *Communications Surveys & Tutorials, IEEE*, 17(1):334–357, 2014.

[22] Andreas Moser, Christopher Kruegel, and Engin Kirda.

Exploring multiple execution paths for malware analysis. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 231–245. IEEE, 2007.

[23] Sebastian Neuner, Victor van der Veen, Martina Lindorfer, Markus Huber, Georg Merzdovnik, Martin Mulazzani, and Edgar Weippl. Enter sandbox: Android sandbox comparison. *arXiv preprint arXiv:1410.7749*, 2014.

[24] Jon Oberheide and Charlie Miller. Dissecting the android bouncer. *SummerCon2012, New York*, 2012.

[25] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage against the virtual machine: Hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*, EuroSec '14, pages 5:1–5:6, New York, NY, USA, 2014. ACM.

[26] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *Proceedings of the 20th Annual Network & Distributed System Security Symposium (NDSS)*, 2014.

[27] John Regehr Raimondas Sasnauskas. Intent fuzzer: Crafting intents of death. 2014.

[28] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting runtime data in android applications for identifying malware and enhancing code analysis. 2015.

[29] Siegfried Rasthofer, Irfan Asrar, Stephan Huber, and Eric Bodden. An investigation of the android/badaccents malware which exploits a new android tapjacking attack. 2015.

[30] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 329–334. ACM, 2013.

[31] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Catch me if you can: Evaluating android anti-malware against transformation attacks. *Information Forensics and Security, IEEE Transactions on*, 9(1):99–108, 2014.

[32] Xiaoyong Zhou Mehool Intwala Apu Kapadia XiaoFeng Wang Roman Schlegel, Kehuan Zhang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *Proceedings of the Network and Distributed System Security Symposium*, 2011.

[33] Snoopwall. Flashlight apps threat assessment report. http://www.snoopwall.com/wp-content/uploads/2014/10/Flashlight-Spyware-Appendix-2014.pdf, 2012. Accessed: 2015-07-14.

[34] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1808–1815. ACM, 2013.

[35] Christian Fritz Eric Bodden Alexandre Bartel Jacques Klein Yves Le Traon Damien Octeau Patrick McDaniel Steven Arzt, Siegfried Rasthofer. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *PLDI 14*, 2014.

[36] Vanja Svajcer. Sophos mobile security threat report. In *Mobile World Congress*, 2014.

[37] Victor van der Veen. Dynamic analysis of android malware. Master's thesis, VU University Amsterdam, 2013.

[38] Amit Vasudevan and Ramesh Yerraballi. Cobra: Fine-grained malware analysis using stealth localized-executions. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15–pp. IEEE, 2006.

[39] Timothy Vidas and Nicolas Christin. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 447–458. ACM, 2014.

[40] Lok-Kwong Yan and Heng Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *USENIX security symposium*, pages 569–584, 2012.