Rage against the virtual machine: hindering dynamic analysis of Android malware

Article · April 2014

DOI: 10.1145/2592791.2592796

5 authors, including:

Thanasis Petsas
Foundation for Research and Technology - Hellas
9 PUBLICATIONS **461** CITATIONS

SEE PROFILE

Sotiris Ioannidis
Technical University of Crete
272 PUBLICATIONS **5,937** CITATIONS

SEE PROFILE

Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware

Thanasis Petsas,* Giannis Voyatzis,* Elias Athanasopoulos,*
Michalis Polychronakis,† Sotiris Ioannidis*
*Institute of Computer Science, Foundation for Research and Technology—Hellas, Greece
†Columbia University, USA
{petsas, jvoyatz, elathan, sotiris}@ics.forth.gr, mikepo@cs.columbia.edu

ABSTRACT

Antivirus companies, mobile application marketplaces, and the security research community, employ techniques based on dynamic code analysis to detect and analyze mobile malware. In this paper, we present a broad range of anti-analysis techniques that malware can employ to evade dynamic analysis in emulated Android environments. Our detection heuristics span three different categories based on (i) static properties, (ii) dynamic sensor information, and (iii) VM-related intricacies of the Android Emulator. To assess the effectiveness of our techniques, we incorporated them in real malware samples and submitted them to publicly available Android dynamic analysis systems, with alarming results. We found *all* tools and services to be vulnerable to most of our evasion techniques. Even trivial techniques, such as checking the value of the IMEI, are enough to evade some of the existing dynamic analysis frameworks. We propose possible countermeasures to improve the resistance of current dynamic analysis tools against evasion attempts.

1. INTRODUCTION

The popularity of Android, in conjunction with the openness of the platform, has made it an attractive target for attackers [13]. The antivirus and research community have responded to this increasing security concern through malicious app analysis services and tools. Google has also created Bouncer [1], a service that automatically scans and detects malicious apps. Scanning an app for inferring its potentially hidden malicious activities can be based on static [22, 25] and dynamic analysis [14, 17, 19, 28]. Unfortunately both static and dynamic analysis approaches can be evaded. As far as static analysis is concerned, researchers have demonstrated a series of techniques which can exploit currently available analysis tools [30]. As we demonstrate in this work, dynamic analysis using emulation for inspecting Android malware is not perfect either. A malicious program can try to infer whether it runs in an emulated environment, and therefore evade detection by pausing all malicious activities.

Specifically, in this paper we investigate how Android applications can infer whether they are running on an emulated `ARM` architecture or on actual hardware. We begin with the creation of a taxonomy of possible ways for identifying features of the execution environment using sets of heuristics. Our heuristics span a wide spectrum of sophistication. Many of them are simple, and can be thwarted by simple modifications to the emulated environment for confusing the heuristic, such as using realistic values for static properties like the serial number of the device. Others are more robust, as the emulated environment needs to incorporate realistic output of mobile sensors, such as the accelerometer. Finally, we present a set of heuristics that require design changes in the actual emulated environment to be defeated.

To assess the importance of our findings we repackaged a set of actual malware samples, by incorporating the developed heuristics, and submitted them to online analysis tools. Surprisingly, all of the tested analysis tools could be evaded using some of our heuristics. There was *no single* malware analysis service that could cope with *all* of the tested heuristics. Furthermore, at least 5 of the 12 analysis tools we checked can be evaded by using heuristics as simple as checking the IMEI value. More complex heuristics based on virtual machine intricacies could evade all but four of the tested services. Those four services do not support native code, and thus can only be used by reviewing a subset of Android apps. Finally, *all* tested services were vulnerable to sensor-based heuristics. We argue that current practices for malware analysis can be easily evaded by demonstrating that actual malware can conceal its malicious functionality from publicly available malware analysis services. We propose a set of countermeasures for making emulated Android app analysis environment more robust to evasion attempts.

2. ANTI-ANALYSIS TECHNIQUES

Anti-analysis techniques that can be employed by Android apps to evade detection can be classified in three categories: (a) *static heuristics*, based on static information always initialized to fixed values in the emulated environment, (b) *dynamic heuristics*, based on observing unrealistic behavior of various sensors, and (c) *hypervisor heuristics*, based on incomplete emulation of the actual hardware. Table 1 provides a summary of all categories, along with some representative examples.

2.1 Static Heuristics

The *static* set includes heuristics that can be used for detecting emulated environments by checking the presence and the content of unique device identifiers, such as the serial number (device ID), the current build version, or the layout of the routing table.

Device ID. Each smartphone contains an IMEI (International Mobile Station Equipment Identity), which is a unique number identifying it in the GSM network. The IMEI has already been used by malicious Android apps to hinder analysis by malware detection tools running on emulators [2]. Another mobile device identifier is

EuroSec'14, April 13 - 16 2014, Amsterdam, Netherlands.

http://dx.doi.org/10.1145/2592791.2592796.

Category	Type	Examples
Static	Pre-initialized static information	IMEI has a fixed value, routing table is fixed
Dynamic	Dynamic information does not change	Sensors produce always the same value
Hypervisor	VM instruction emulation	Native code executes differently

Table 1: A summary of the main types of VM detection heuristics that can be used by mobile malware, along with representative examples.

the IMSI (International Mobile Subscriber Identity), which is associated with the SIM card found in the phone. Our simplest evasion heuristics are based on checking these identifiers, e.g., whether the IMEI is equal to `null`, which is true for the default configuration of Android Emulator. We will refer to this kind of heuristics using the abbreviation **idH**.

Current build. Another way to identify emulated environments is by inspecting information related to the current build, as extracted from system properties. For instance, the Android SDK provides the public class `Build`, which contains fields such as `PRODUCT`, `MODEL`, and `HARDWARE`, that can be examined in order to detect if an application is running on an emulator. For example, a default Android image on an emulator has the `PRODUCT` and `MODEL` fields set to `google_sdk`, and the `HARDWARE` field set to `goldfish`. We have implemented a number of heuristics based on this kind of checks, to which we refer as **buildH**.

Routing table. An emulated Android device by default runs behind a virtual router within the 10.0.2/24 address space, isolated from the host machine's network. The emulated network interface is configured with the IP address 10.0.2.15. The configured gateway and DNS servers have also specific values. We use these networking properties as another detection heuristic. Specifically, the heuristic checks listening sockets and established connections (through `/proc/net/tcp`), and attempts to find a port number associated with addresses 10.0.2.15 and 0.0.0.0, as an indication of an emulated environment. We refer to this heuristic as **netH**.

2.2 Dynamic Heuristics

Mobile phones are equipped with a variety of sensors, including an accelerometer, gyroscope, GPS, gravity sensor, etc. Essentially, these sensors output values based on information collected from the environment, and therefore simulating them realistically is a challenging task. The existence of sensors is a key difference between smartphones and conventional computing systems. The increasing number of sensors on smartphones presents new opportunities for the identification of actual mobile devices, and thus for the differentiation and detection of emulators. For instance, there are studies focused on smartphone fingerprinting based on sensor flaws and imperfections [3, 20]. Such fingerprinting approaches can be leveraged for the detection of emulated environments.

By default, the Android emulator cannot simulate device movements; this can be achieved through additional sensor simulators [4]. Current builds of the Android Emulator also support partially or not at all simulation of other types of sensors. In our testing of the available simulated sensors, we found that they generated the same value at equal time intervals equal in average to 0.8 second with negligible standard deviation (equals to 0.003043). The CDF of the intervals between accelerometers' events as observed in an Android Emulator running for a couple of minutes is shown in Figure 1. We found that the CDF for the rest of the sensors in Android Emulator follows a similar pattern. We implemented our sensor-based heuristics by taking advantage of the `SensorManager` [5] class of the Android API. We developed an Android Activity that attempts to register a sensor listener to monitor its output values using the following approach. First, we try to register a sensor listener. If the registration fails, then the execution probably takes place in an emulated environment (except in the case of an actual device that does not support the specific sensor). Otherwise, if sensor registration is successful, then we check the `onSensorChanged` callback method, which is called when sensor values change. If the sensor values or time intervals observed are the same between consecutive calls of this method, then we assume that the app is running on an emulated environment and we unregister the sensor listener. We implemented this sensor-based heuristic for the accelerometer (**accelH**), magnetic field (**magnFH**), rotation vector (**rotVecH**), proximity (**proximH**), and gyroscope (**gyrosH**) sensors.

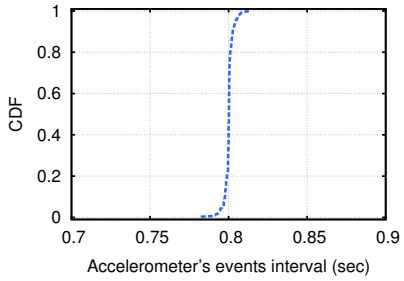

Figure 1: The CDF of the intervals between accelerometers' events as observed in an Android Emulator running for a few minutes.

2.3 Hypervisor Heuristics

Identifying QEMU scheduling. Our first hypervisor heuristic is related to QEMU scheduling [26], and the fact that QEMU does not update the virtual program counter (PC) at every instruction execution for performance reasons. As translated instructions are executed natively, and increasing the *virtual* PC needs an additional instruction, it is sufficient and faster to increase the virtual PC only when executing instructions that break linear execution, such as branch instructions. This means that if a scheduling event occurs during the execution of a basic block, it would be impossible for the virtual PC to be reconstructed. For this reason, scheduling in a QEMU environment happens only *after* the execution of a basic block, and never within its execution.

A proof of concept QEMU Binary Translation (BT) detection technique based on a histogram of the scheduling addresses of a thread has already been implemented [26]. In a non-emulated environment, a large set of various scheduling points will be observed, as scheduling can happen at an arbitrary time, whereas in an emulated environment, only a specific scheduling point is expected to be seen, at the beginning of a basic block, as scheduling happens after the execution of a complete basic block. We have implemented this technique and used it in our experiments as an extra heuristic, abbreviated as **BTdetectH**. In Figure 2, we show the different behaviors of scheduling points by running this detection heuristic on an Android Emulator and on a real device.

Identifying QEMU execution using self-modifying code. As a second heuristic, we implemented a novel QEMU detection technique (we call **xFlowH**) based on the fact that QEMU tracks modi-

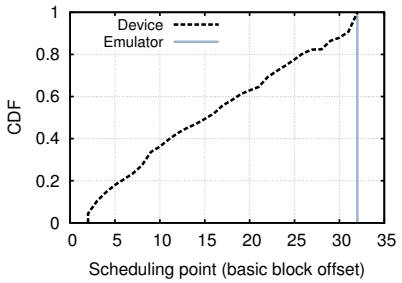

Figure 2: Due to optimizations, QEMU does not update the virtual PC on every instruction execution, and therefore many of the scheduling events that can take place are not exhibited on an emulated environment.

fications on code pages.[1] The technique is based on incurring variations in execution flows between an emulator and a real device through self-modifying code.

ARM processors include two different caches, one for instructions accesses (I-Cache) and one for data accesses (D-Cache) [18]. Harvard architectures (like ARM) do not ensure coherence between I-Cache and D-Cache. Therefore, the CPU may execute an old (possibly invalid) piece of code after a newly one has already been written in main memory. This issue can be resolved by enforcing consistency between the two caches, which can be achieved through two operations: (a) *cleaning* main memory, so as the newly written code lying in the D-Cache to be moved into main memory; and (b) invalidating the I-Cache so that it can be repopulated with the new content of the main memory. In native Android code, this can be done through the `cacheflush` function, which carries out the above operations through a system call.

We implemented an example of self-modifying (native) code that uses a memory segment with write and execute rights which is overwritten several times, in a loop, with the content (code) of two different functions, `f1` and `f2`, alternately. After each code patch, we run the code of this segment, which in turn runs either `f1` or `f2`. These are two simple functions which both append their name at the end of a global string variable, so that the function call sequence can be inferred. To achieve an alternating call sequence, we have to synchronize the caches through a `cacheflush` call as described previously.

We ran this code, along with the extra calls for cache synchronization after each patch, on a mobile device and on the emulator, with the same results—each execution produced a consistent function call sequence as determined by the loop. Then, we performed the same experiment, but this time excluded the `cacheflush` calls. As expected, on the mobile device we observed a random call sequence for each run. As the caches are not synchronized before each call, the I-Cache may contain stale instructions because it is not explicitly invalidated. Interestingly, we found that this does not happen on the emulator. Instead, the call sequence was exactly the same as in the first case, when the caches were consistent before each function call. This behavior is expected, as QEMU discards its translated block for the previous version of the code, and retranslates the newly generated code, as it tracks modifications on code pages and ensures that the generated code always matches the target instructions in memory [16].

[1] A similar heuristic has been developed independently and concurrently with this work [6]. At the time of writing, that work does not include any evaluation of the heuristic with real-world analysis tools and services.

Family	Package name	Heuristic	Description
BadNews	ru.blogspot. playsib.savageknife	magnFH	Data extrusion
BaseBridge	com.keji.unclear	accelH	Root exploit
Bgserv	com.android. vending.sectool.v1	netH	Bot activity
DroidDream	com.droiddream. bowlingtime	gyrosH	Root expoit
DroidKungFu	com.atools.cuttherope	rotVecH	Root exploit
FakeSMS Installer	net.mwkekdsf	proximH	SMS trojan
Geinimi	com.sgg.sp	buildH	Bot avtivity
Zsone	com.mj.iCalendar	idH	SMS trojan
JiFake	android.packageinstaller	BTdetectH	SMS trojan
Fakemart	com.android.blackmarket	xFlowH	SMS trojan

Table 2: Malware samples used for our study.

3. IMPLEMENTATION

We have implemented the heuristics described in Section 2 using the Android SDK. For BTdetectH and xFlowH, we used the Java Native Interface (JNI) to invoke the native code that implements the functionality of each heuristic. We developed a simple Android application (test app) that runs our heuristics in the background, and for each one collects information about its effectiveness. The collected data is sent to an HTTP server to be stored in a local database. Moreover, we incorporated our heuristics in a set of well known Android malicious apps. For this purpose, we used Smali/Baksmali [7] along with Apktool [8], which we used for disassembling and reassembling process. The incorporation of our heuristics in malicious apps was done by patching the Smali Dalvik bytecode (generated by the disassembly process) with the Smali code of each heuristic, which was previously extracted from our developed test app. Each malicious app was modified to carry one of the implemented heuristics, as listed in Table 2. At first, we ran each original sample as is, both on the emulator and on a real device, and observed through the `logcat` command of Android logging system the initial spawned Android activities and services. Afterwards, we patched these components with one of the heuristics, which, depending on the detection result, decides whether to continue the execution of the component or not.

We tested the repackaged applications both on multiple emulators and actual devices to make sure that the malicious behavior is triggered only when the execution happens on a real device. Note that apart from the above changes in the produced code of malicious apps, no other additions are needed in any other parts of the APK files, except for the following cases. The idH heuristic requires the `READ_PHONE_STATE` permission explicitly declared in the Android Manifest file, to be able to retrieve information about the phone state. For the BTdetectH and xFlowH heuristics, a new folder named `lib` needs to be created inside the top level directory containing the desired native code in the form of shared libraries.

4. EXPERIMENTAL EVALUATION

In this section, we present the results of our evaluation regarding the effectiveness of the heuristics we presented in Section 2. Each heuristic was added in real malware and analyzed with various dynamic analysis services and tools. For each case, we record which of the heuristics managed to detect the emulated environment and which failed. We first describe the malware dataset and the dynamic analysis services we used, then proceed with a summary of our methodology, and finally present and discuss our findings.

4.1 Data and Tools

Malware Samples. We patched a number of well known Android malicious apps with the code of our detection techniques using the process described in Section 3. We used 10 samples from different malware families with distinct capabilities, including root ex-

Type	Tool	Web Page
Offline	DroidBox	http://code.google.com/p/droidbox/
	DroidScope	http://code.google.com/p/decaf-platform/wiki/DroidScope
	TaintDroid	http://appanalysis.org/
Online	Andrubis	http://anubis.iseclab.org/
	SandDroid	http://sanddroid.xjtu.edu.cn/
	ApkScan	http://apkscan.nviso.be/
	VisualThreat	http://www.visualthreat.com/
	Tracedroid	http://tracedroid.few.vu.nl/
	CopperDroid	http://copperdroid.isg.rhul.ac.uk/copperdroid/
	APK Analyzer	http://www.apk-analyzer.net/
	ForeSafe	http://www.foresafe.com/
	Mobile Sandbox	http://mobilesandbox.org/

Table 3: Android malware analysis tools and services used in our evaluation.

	idH	buildH	netH	accelH	magnFH	rotVecH	proximH	gyrosH	BTdetectH	xFlowH
DroidBox	✓	✗	✗	✗	✗	✗	✗	✗	*JNI NS*	*JNI NS*
DroidScope	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
TaintDroid	✗	✗	✗	✗	✗	✗	✗	✗	*JNI NS*	*JNI NS*
Andrubis	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
SandDroid	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
ApkScan	✓	✗	✗	✗	✗	✗	✗	✗	*JNI NS*	*JNI NS*
VisualThreat	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Tracedroid	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
CopperDroid	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Apk Analyzer	✓	✓	✓	✗	✗	✗	✗	✗	*JNI NS*	*JNI NS*
ForeSafe	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Mobile Sandbox	✓	✗	✗	✗	✗	✗	✗	✗	*JNI NS*	*JNI NS*

Table 4: Resilience of Android dynamic analysis tools against VM evasion techniques. "✓" indicates *resilient*, "✗" indicates *vulnerable* and "*JNI NS*" indicates lack of support for JNI.

ploits, sensitive information leakage, SMS trojans, and so on. All tested samples are publicly available and are part of the Contagio Minidump [9]. Table 2 provides a summary of the set of malware samples used, along with the heuristics used in each case.

Dynamic Analysis Services. The Android dynamic analysis services and tools used in our evaluation are listed in Table 3. We used both standalone tools available for download and local use, as well as online tools which analyze submitted samples online.

We used three popular open source Android app analysis tools: DroidBox [10], DroidScope [35], and TaintDroid [21]. All three tools execute Android applications in a virualized environment and produce analysis reports. DroidBox offers information about about incoming/outgoing traffic, read/write operations, services invoked, circumvented permissions, SMS sent, phone calls made, etc. DroidScope performs API-level as well as OS-level profiling of Android apps and provides insight about information leakage. TaintDroid is capable of performing system-wide information flow tracking from multiple sources of sensitive data in an efficient way.

In addition to standalone tools, we also used publicly available online services that dynamically analyze Android apps, briefly described below. Andrubis [14] performs both static and dynamic analysis on unwanted Android apps. SandDroid performs permission/component analysis as well as malware detection/classification analysis. ApkScan provides information including file accesses, network connections, phone calls, SMS sent, information leakage, and cryptographic activities. VisualThreat provides information spanning from network activity and data leakage to malware family detection through API correlation. TraceDroid emulates some actions, when analyzing apps, such as user interaction, incoming calls, SMS messages, which can reveal malicious intents. CopperDroid [31] is built on top of QEMU and performs out-of-the-box dynamic analysis (both OS-level and API-level) of Android malware. APK Analyzer, ForeSafe and Mobile Sandbox [33] are also dynamic analysis services, which inspect mobile malware by following similar approaches as the previously mentioned tools.

4.2 Methodology

To evaluate the effectiveness of our heuristics, we follow two different approaches. In the first approach we submit an application sample that incorporates *all* of the heuristics, as described in Section 3, to each online service of our set. During the analysis, the application tests the effectiveness of the heuristics by trying them all sequentially for inferring if the execution environment is emulated or not. For each heuristic tried, the app communicates the result to an HTTP server. After that, we are able to ascertain which of the heuristics were effective on identifying a virtualized environment and which were not.

In order to perform a more realistic analysis and demonstrate how such heuristics can complicate the work of various dynamic-analysis tools from drawing correct conclusions, we follow a different approach. We modify actual malicious apps so as to incorporate one of the heuristics, following the process described in Section 3, and we submit both the original sample and the repackaged one to each service. Then, we collect the generated reports and compare them to find differences in the observed behaviors.

4.3 Evasion Results

The results of our evaluation are summarized in Table 4. The first three rows correspond to standalone analysis tools, and the rest to online analysis services. We next describe our key findings based on the collected results.

Finding 1. *All the studied dynamic analysis tools and services are vulnerable to five or more heuristics.* As we can see in Table 4, all the tools we tested can be easily evaded using just one of the proposed heuristics. Specifically, only 6 out of the 12 tools are resilient to our simplest static heuristic **idH**, which checks the IMEI of the device. We also found that only one analysis service, Apk Analyzer, is resilient to all of the static heuristics. In addition, all of the services are vulnerable to all of the dynamic heuristics, which are based on using the sensors of the device, as well as the hypervisor heuristics that take advantage of QEMU code generation intricacies. Note that four tools (DroidBox, TaintDroid, ApkScan, Apk Analyzer) do not support the execution of native code at all, which means that we could not test the effectiveness of the hypervisor heuristics on them. We were able to infer that through the exception messages thrown when our test app attempted to load native code through JNI. All these messages were included in the HTTP request carrying the results.

Finding 2. *All analysis tools failed to correctly infer the malicious behavior of the repackaged malware samples.* We compared all reports produced by all of our studied tools manually (both offline and online ones) following our second evaluation approach, and observed that the results are consistent with those found by our first methodology. That is, the repackaged samples integrating a specific heuristic managed to hide their malicious behavior successfully, and went unnoticed from the tools previously found vulnerable to the same heuristic. We were able to validate this from the reduced number of the observed incidents and activities, as well as from the reduced network traffic observed.

Finding 3. *All of the studied online analysis services can be fingerprinted based on inferred information about their execution environment*. During our analysis, we did not receive any requests with results from three of the online analysis services (CopperDroid, Vi-

sual Threat, and APK Analyzer) when following the first approach. Apparently, those tools analyze apps in an environment with network connectivity disabled. Nonetheless, we observed that if an application attempted to write a file during the analysis process then the filename was reflected on the results page of all these tools. Thus, we used this trick in order to test the effectiveness of our heuristics by creating files with special filenames, which indicate if a particular heuristic was successful or not. So, by using this approach, we found that we are able to infer information about the environment, i.e., the Build information, which can be further used for services fingerprinting.

Finding 4. *Only one of the studied tools provides information about VM evasion attempts.* We found, by analyzing the reports generated by each analysis tool, that only Apk Analyzer has the feature of detecting evasion behavior in the submitted samples, and generates relevant alerts in the generated reports (for example we observed the message: *"May tried to detect the virtual machine to hinder analysis"*). Moreover, it reports the ways used to find the evasion behavior. For example for the repackaged samples containing the idH and buildH heuristics, it mentions the VM artifact strings found in memory during the execution of the sample. Moreover, the netH heuristic was also not effective on this service as the default IP address of the network interface used on Android Emulator was modified from 10.0.2.15 to be 192.168.0.24. Nonetheless, our netH heuristic can be improved if it is changed so as to search for an arbitrary IP address and the 0.0.0.0 one to be associated with the ADB port in `/proc/net/tcp` interface.

Although Apk Analyzer was found resilient to all of our static heuristics, this was not the case for the dynamic heuristics. Indeed, we found that using at least one dynamic heuristic, i.e., accelH, into a repackaged sample is sufficient in order to hide its behavior from Apk Analyzer. Furthermore, we observed that for all the repackaged samples equipped with dynamic heuristics, Apk Analyzer did not produce alerts related with VM detection attempts.

5. COUNTERMEASURES

Mobile malware with VM detection capabilities can pose a significant threat, since it is possible to evade detection and therefore nullify the work of dynamic analysis tools that run on emulators. Moreover, Google bouncer, which is the official tool used in order to detect malicious applications that are about to be published in Google Play, is based on dynamic analysis conducted in the default emulated environment of Android on top of QEMU. Google Bouncer is also vulnerable to environment detection techniques [11]. In this section we propose a set of defenses that can be applied in the current emulated environment of Android so as to make it more realistic.

Emulator Modifications. The Android Emulator can be modified easily in order to be resistant to our proposed static heuristics. Mobile device identifiers such as IMEI and IMSI which are used from our idH heuristic are all configurable in Android emulator. By looking at the Telephony Manager service in the Android Emulator's source code and through code analysis, one can find the place where the modem device is emulated, which is implemented as part of QEMU [12]. Thus the IMEI, IMSI, as well as other features can be modified to make the emulator resemble to a real device. The buildH heuristic that uses the information of the current build can be easily deceived by modifying the build properties loaded by Android Emulator. These properties are defined in the `build.prop` file of Android Emulator's source code. Finally, the default network properties of Android Emulator can be modified to provide protection against netH as Apk Analyzer does.

Realistic Sensor Event Simulation. Our second set of heuristics, the dynamic heuristics, are based on the variety of sensors that a mobile device supports. As already mentioned, Android Emulator supports trivially detectable sensor simulation, with sensor events occurring at precise intervals and the produced values not changing between consecutive events. All the dynamic heuristics (accelH, magnFH, rotVecH, proximH and gyrosH) are based on this behavior. In order to make all dynamic heuristics ineffective, better sensor simulation is required. Nonetheless, realistic simulation of such hardware components is challenging and requires in-depth knowledge of both the range of values that these devices can produce, as well as realistic user interaction patterns. In this context, external software simulators [4] could be used or record-and-replay approaches [23], in order to simulate sensor data at real time as an additional component of the Android Emulator.

Accurate Binary Translation. Binary translation used by QEMU, on which the BTdetectH heuristic is based, is an essential operation of the Android Emulator, in which each `ARM` instruction is translated into the `x86` equivalent in order to be able to run on the `x86`-based host machine. BTdetectH is based on a fundamental operation of the Android Emulator, as already discussed in Section 2, and is not trivial to change it. One way to remedy this issue is by making the binary translation process of QEMU more accurate to the real execution used in the CPU of a device. That is, the virtual program counter has to be always updated after an instruction, as happens when instructions are getting executed in a CPU. Thus, this requires revision and expansion of the current binary translation operation in QEMU. On the other hand, this approach would end up producing a higher execution overhead, making QEMU, and consequently the Android Emulator, easily detectable (e.g., by comparing the different execution times that could be observed in an emulator and in a real device for specific operations).

Hardware-Assisted Virtualization. Another way to cope with the above issue is to use hardware-assisted virtualization, which is based on architectural support that facilitates building a virtual machine monitor and allows guest OSes to run in isolation. For example, the upcoming hardware-assisted `ARM` virtualization technology [15] can be used to avoid the process of binary translation and the problems associated with it. By replacing instruction emulation (QEMU) with such technology, BTdetectH and xFlowH heuristics which are based on VM intricacies would become ineffective.

Hybrid Application Execution. Furthermore, another solution to our hypervisor heuristics would be to use real mobile devices to execute applications that contain native code. Both these two heuristics (BTdetectH and xFlowH) require native code execution in order to act. Hybrid application execution would be the most secure and efficient way for a dynamic analysis tool against all the suggested evasion heuristics. That is, application bytecode can run in a patched version of Android Emulator shielded with all protection measures described above; when an application is attempting to load and run native code, then the execution of the native code can be forwarded and take place on a real device.

6. RELATED WORK

Rastogi et al. [30], evaluated the top commercial anti-malware products available for Android devices against various obfuscation techniques. Based on their results, all tested products were found to be vulnerable to common obfuscation techniques. Their study was based on *static analysis* tools. In contrast, we followed a similar approach to evaluate *dynamic analysis* tools for Android. Sawar et al. [32], argue that dynamic taint tracking is ineffective for the detection of privacy leaks in malicious Android apps, and implemented a set of attacks against TaintDroid that a malware could

use remove the generated taint marks. In our study, we also used TaintDroid to evaluate the effectiveness of our heuristics.

There are numerous studies that focus on VM evasion in conventional systems. Raffetseder et al. [29] present a number of techniques for detecting system emulators. Their techniques make use of various features, such as specific CPU bugs, model-specific registers (MSRs), instruction length limits, relative performance measurements and many others. Willems et al. [34] present a different approach by introducing code sequences that have an implicitly different behavior on a native machine when compared with an emulator. These techniques are based on side-effects of the particular operations and imperfections in the emulation process, (e.g., applications can follow a different control flow path if they are emulated on architectures that support self-modifying code), and are very similar to our approach used in the xFlowH heuristic.

Paleari et al. [27], present an automatic way of producing *red-pills*, that is pieces of code capable to detect whether they are executed in a CPU emulator or in a physical CPU. Their approach was implemented in a prototype, which was used for discovering new red-pills, involving hundreds of different opcodes, for two popular CPU emulators: QEMU and BOCHS. Lindorfer et al. [24], introduce DISARM, a system for detecting environment-sensitive malware. A malware sample is executed in different analysis sandboxes, and the observed behaviors from each sandbox are compared in order to detect evasive malwaring. All these studies propose techniques that can be used by malware for detecting x86 virtualized environments. Our study proposes similar techniques that can be used by mobile malware and particularly by malicious apps targeting Android (mostly related to the ARM architecture).

7. CONCLUSION

In this paper, we explored how dynamic analysis can be evaded by malicious apps targeting the Android platform. We implemented and tested heuristics of increasing sophistication by incorporating them in actual malware samples, which attempt to hide their presence when analyzed in an emulated environment. We tested all re-packaged malware samples with standalone analysis tools and publicly available scanning services, and monitored their behavior. There was no service or tool that could not be evaded by at least some of the tested heuristics. The work we conducted raises important questions about the effectiveness of existing analysis systems for Android malware. For this reason, we proposed a number of possible countermeasures for improving the resistance of dynamic analysis tools for Android malware against VM detection evasion.

Acknowledgements

This work was supported in part by the FP7 projects NECOMA, OPTET, SysSec and by the FP7-PEOPLE-2010-IOF project XHUNTER, funded by the European Commission under Grant Agreements No. 608533, No. 317631, No. 254116 and No. 273765.

8. REFERENCES

[1] http://googlemobile.blogspot.com/2012/02/android-and-security.html.

[2] http://vrt-blog.snort.org/2013/04/changing-imei-provider-model-and-phone.html.

[3] http://blog.sfgate.com/techchron/2013/10/10/stanford-researchers-discover-alarming-method-for-phone-tracking-fingerprinting-through-sensor-flaws/.

[4] http://code.google.com/p/openintents/wiki/SensorSimulator.

[5] http://developer.android.com/reference/android/hardware/SensorManager.html.

[6] https://bluebox.com/corporate-blog/android-emulator-detection/.

[7] http://code.google.com/p/smali/.

[8] http://code.google.com/p/android-apktool/.

[9] http://contagiominidump.blogspot.com/.

[10] http://code.google.com/p/droidbox/.

[11] https://www.duosecurity.com/blog/dissecting-androids-bouncer.

[12] https://codepainters.wordpress.com/2009/12/11/android-imei-number-and-the-emulator/.

[13] 99% of all mobile threats target Android devices. http://www.kaspersky.com/about/news/virus/2013/99_of_all_mobile_threats_target_Android_devices.

[14] Anubis/Andrubis: Analyzing Unknown Binaries. http://anubis.iseclab.org/.

[15] Arm: Virtualization extensions. http://www.arm.com/products/processors/technologies/virtualization-extensions.php.

[16] QEMU Internals. http://ellcc.org/ellcc/share/doc/qemu/qemu-tech.html.

[17] T. Bläsing, A.-D. Schmidt, L. Batyuk, S. A. Camtepe, and S. Albayrak. An android application sandbox system for suspicious software detection. In *MALWARE*, 2010.

[18] Bramley Jacob. Caches and Self-Modifying Code. http://community.arm.com/groups/processors/blog/2010/02/17/caches-and-self-modifying-code.

[19] J. Calvet, J. M. Fernandez, and J.-Y. Marion. Aligot: cryptographic function identification in obfuscated binary programs. In *CCS*, 2012.

[20] S. Dey, N. Roy, W. Xu, and S. Nelakuditi. Acm hotmobile 2013 poster: Leveraging imperfections of sensors for fingerprinting smartphones. *SIGMOBILE Mob. Comput. Commun. Rev.*, 17(3), Nov. 2013.

[21] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, 2010.

[22] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *USENIX Security,*, 2011.

[23] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. Reran: Timing- and touch-sensitive record and replay for android. In *ICSE*, 2013.

[24] M. Lindorfer, C. Kolbitsch, and P. Milani Comparetti. Detecting environment-sensitive malware. In *RAID*, 2011.

[25] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: Statically Vetting Android apps for Component Hijacking Vulnerabilities. In *CCS*, 2012.

[26] F. Matenaar and P. Schulz. Detecting Android Sandboxes. http://www.dexlabs.org/blog/btdetect.

[27] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi. A fistful of red-pills: how to automatically generate procedures to detect cpu emulators. In *WOOT*, 2009.

[28] T. H. Project. Android Reverse Engineering (A.R.E.) Virtual Machine. http://www.honeynet.org/node/783.

[29] T. Raffetseder, C. Kruegel, and E. Kirda. Detecting system emulators. In *ISC*, 2007.

[30] V. Rastogi, Y. Chen, and X. Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *ASIA CCS*, 2013.

[31] A. Reina, A. Fattori, and L. Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In *EUROSEC*, 2013.

[32] G. Sarwar, O. Mehani, R. Boreli, and D. Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *SECRYPT*, 2013.

[33] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, and J. Hoffmann. Mobile-sandbox: Having a deeper look into android applications. In *SAC*, 2013.

[34] C. Willems, R. Hund, A. Fobian, D. Felsch, T. Holz, and A. Vasudevan. Down to the bare metal: Using processor features for binary analysis. In *ACSAC '12*, 2012.

[35] L. K. Yan and H. Yin. DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In *USENIX Security*, 2012.