Evading Android Runtime Analysis via Sandbox Detection

Timothy Vidas
Carnegie Mellon University
tvidas@cmu.edu

Nicolas Christin
Carnegie Mellon University
nicolasc@andrew.cmu.edu

ABSTRACT

The large amounts of malware, and its diversity, have made it necessary for the security community to use automated dynamic analysis systems. These systems often rely on virtualization or emulation, and have recently started to be available to process mobile malware. Conversely, malware authors seek to detect such systems and evade analysis. In this paper, we present techniques for detecting Android runtime analysis systems. Our techniques are classified into four broad classes showing the ability to detect systems based on differences in behavior, performance, hardware and software components, and those resulting from analysis system design choices. We also evaluate our techniques against current publicly accessible systems, all of which are easily identified and can therefore be hindered by a motivated adversary. Our results show some fundamental limitations in the viability of dynamic mobile malware analysis platforms purely based on virtualization.

Categories and Subject Descriptors

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

Keywords

Evasion; malware; android; security; sandbox

1. INTRODUCTION

In the past couple of years, mobile devices have become sophisticated computing environments with increased computing power and network connectivity. This has made them considerably closer to traditional computing platforms such as PCs, than to the telephones they were initially designed to replace. As a result of this trend, security threats that traditionally only applied to PCs can now also be encountered on mobile devices. In particular, as miscreants discover that mobile devices can be targeted and exploited for financial gain, new forms of malware are created to do so. Some mobile malware is designed to work in tandem with PC-oriented malware while other focuses on mobile devices exclusively [18,36,40].

When a new piece of malware is discovered, it must be analyzed in order to understand its capabilities and the threat it represents. One popular method of analysis, *dynamic analysis*, consists of executing the malware in a controlled environment to observe effects to the host system and the network.

ASIA CCS'14, June 4–6, 2014, Kyoto, Japan.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2800-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2590296.2590325.

On traditional PCs, the controlled environment used for dynamic analysis is often created through host virtualization. PC malware authors have consequently taken to design malware that can detect virtual environments [19,24,30,34], and exhibit alternative, benign behavior, rather than the malicious behavior which would occur on actual hardware. This kind of evasion allows malware to thwart dynamic analysis systems, and has fueled an arms race between those improving the realism of virtualized environments and those wishing to detect them.

However, virtualization technology has considerably matured in the past few years, and a number of users have migrated physical machines to virtual instances. Today, production services may be run in corporate virtualization "farms" or even in space rented in a "cloud." As a result, virtualized environments are not merely used for sandboxing malware anymore, but have become commonplace for a number of applications. In turn, the ability for malware to detect a virtual environment is losing its usefulness, as virtualization is no longer a near-certain indication that dynamic malware analysis is taking place.

On the other hand, there are, so far, only limited use cases for virtual environments on mobile devices. Likewise, emulated environments are typically only used by developers or dynamic analysis systems. For this reason, mobile malware authors may still employ virtualization or emulation detection to alter behavior and ultimately evade analysis or identification.

In this paper we contribute several techniques that can be used to detect a runtime analysis in Android. Some of these techniques are specific to the Android framework or a virtualized environment (e.g., emulator), and could be quite easily thwarted, while others, based for instance on resource availability, are more general, and much harder to defend against. All of the techniques we present require minimal or no privileges, and could be invoked from typical applications found in online marketplaces.

The primary contribution of this paper is thus to demonstrate that dynamic analysis platforms for mobile malware that purely rely on emulation or virtualization face fundamental limitations that may make evasion possible.

The rest of this paper is organized as follows. We start by presenting related work in Section 2. We then propose different emulation detection techniques in Section 3. In Section 4 we address an obvious countermeasure to some detection techniques by demonstrating alternate implementations of the methods described in Section 3. We evaluate how current systems fare against the proposed techniques in Section 5 before discussion results in Section 6, and concluding in Section 7.

2. RELATED WORK

Automated runtime analysis systems are a popular method of processing large volumes of malware, or as an alternative to having skilled personnel perform lengthy manual analysis. For desktop operating systems there are numerous free [12] and commercial [39] systems that perform such analysis. More recently, a few systems [1, 2, 3, 4, 10, 13, 25] have been proposed for dynamic analysis of mobile malware. Because of their novelty, these systems remain less mature than their desktop counterparts.

PC malware writers responded to the advent of automated systems by finding creative solutions to detect whether code was running in virtualized environments. Virtualization detection routines have been implemented in numerous ways from detecting a vendor-specific API, such as VMWare's communication channel [24], to observing environmental effects of a single CPU instruction [34].

Upon detection, malware may exhibit alternate behavior, hiding its true purpose. This runtime change in control flow results in a different execution path through the malicious software. In 2008, Chen et al. observed that PC malware exhibited reduced behavior in nearly 40% of unique samples when run in debugged environment, and 4% of samples when run in a virtualized environment. However, the samples that exhibit alternate behavior due do environment accounted for 90% of attacks during particular time periods [15]. Moser et al. propose a system for analyzing multiple execution paths by presenting different input and environments to subsequent executions of the same malware [26]. While this system investigates different execution paths, virtualization is not one of the changes used to effect a different execution.

The detection of analysis environments is not limited to detecting the virtualization component itself. Holz et al. describe numerous techniques for detecting honeypots [22]. Some techniques are indeed rooted in virtualization detection, but others focus on other environmental components such as the presence of a debugger.

The concept of evasion has also been subject to a considerable amount of research in the context of network intrusion detection systems (NIDS). Handley et al. observe that a skilled attacker can "evade [network] detection by exploiting ambiguities in the traffic stream" [21]. These ambiguities are classified into three categories: incomplete system implementation, incomplete knowledge of the traffic recipient's system, or incomplete network topology knowledge. Numerous others describe evasion attacks due inconsistencies between the NIDS and the victim computer [14, 16, 20, 27, 32].

The most closely related work to ours offers emulator detection techniques for the PC [33]. Raffetseder et al. detail some "general" detection methods and well as a few that specifically detect the Intel x86 version of the QEMU emulator. At a high level, we explore similar general detection techniques targeting the Android emulator. There is scant similar academic research in evading mobile malware analysis; however there have a been a few industry presentations [28, 31] that look at evasion particular to Google Bouncer. Both of these presentations address some API related detections (which we generalize in Section 3.1) as well as detections that specifically target Bouncer, such as source IP addresses associated with Google. Our work here is more general, in that it tries to pinpoint more fundamental limitations in dynamic analysis on mobile devices. A more general industry presentation by Strazzere explores several Android API-based detections as well as a specific QEMU detection and is complimentary to our research [35].

3. EMULATOR DETECTION

Fundamentally, the concept of emulator detection is rooted in the fact that complete system emulation is an arduous task. By discovering or observing differences between virtual and physical execution an attacker can create software virtualization checks that can be used to alter overall program behavior. Such differences may be an artifact of hardware state not correctly implemented in the virtual CPU, hardware or software components that have yet to be virtualized, or observable execution duration.

In this section we detail several virtualization detection techniques and discuss the results of experimental evaluation of these techniques. The techniques require few or no permissions and work on commodity devices in the standard consumer software configuration. As with any consumer Android device, applications are governed by Linux access control and are thus limited to user-mode processor execution (e.g. devices are not "rooted" or "jailbroken").

We evaluate the detection techniques using emulator instances on similar Windows, Mac, and Linux hosts (each with an i7 processor, 8 GB RAM) as well as six physical devices from major U.S. cellular carriers. We divide detection techniques into the following categories: differences in behavior, performance, hardware and software components, and those resulting from system design choices. For ease in comparison, the first three roughly coincide with existing work in PC emulator detection [33].

3.1 Differences in behavior

Since virtualization is often defined in terms of execution being indistinguishable from that of a real machine, in some sense, any virtualization detection technique can be perceived as a difference in behavior. However, here we focus on behavioral differences specific to software state and independent of system performance.

Detecting emulation through the Android API.

The Android API provides an abstract interface for application programmers. Since many Android devices are smartphones, the API provides a rich interface for telephony operations as well as methods for interacting with other hardware and local storage.

Table 1 enumerates several API methods that return particular values when used with an emulated device. Each of the API-value pairs in Table 1 can be used to explicitly detect an emulated device or used in conjunction with other values in order to determine a likelihood. For example, if the `TelephonyManager.getDeviceId()` API returns all 0's, the instance in question is certainly an emulator because no physical device would yield this value.

Similarly, emulator instances adopt a telephone number based on the Android Debug Bridge (ADB) port in use by the emulator. When an emulator instance starts, the emulator reserves a pair of TCP ports starting with 5554/5555 (a second instance would acquire 5556/5558) for debugging purposes. The adopted telephone number is based on the reserved ports such that the initial emulator instance adopts precisely 1-555-521-5554. Therefore, if the `TelephonyManager.getLine1Number()` API indicates that the device phone number is in the form 155552155xx, then the device is certainly an emulator. Such a number would never naturally be used on a device because the 555 area code is reserved for directory assistance [9]. The presence of the 555 area code may also be used in other emulator detections such as the pre-configured number for voicemail.

Other values in Table 1 are certainly used by the emulator but may also be used by some real devices. Consider the Mobile Country Code (MCC) and Mobile Network Code (MNC) values obtained via the `TelephonyManager.getNetworkOperator()` method. The emulator always returns values associated with T-Mobile USA. Since there are certainly real devices that use the same codes, checks based on the MCC and MNC need to be

API method	Value	meaning
Build.ABI	armeabi	is likely emulator
Build.ABI2	unknown	is likely emulator
Build.BOARD	unknown	is emulator
Build.BRAND	generic	is emulator
Build.DEVICE	generic	is emulator
Build.FINGERPRINT	generic††	is emulator
Build.HARDWARE	goldfish	is emulator
Build.HOST	android-test††	is likely emulator
Build.ID	FRF91	is emulator
Build.MANUFACTURER	unknown	is emulator
Build.MODEL	sdk	is emulator
Build.PRODUCT	sdk	is emulator
Build.RADIO	unknown	is emulator
Build.SERIAL	null	is emulator
Build.TAGS	test-keys	is emulator
Build.USER	android-build	is emulator
TelephonyManager.getDeviceId()	All 0's	is emulator
TelephonyManager.getLine1 Number()	155552155xx†	is emulator
TelephonyManager.getNetworkCountryIso()	us	possibly emulator
TelephonyManager.getNetworkType()	3	possibly emulator (EDGE)
TelephonyManager.getNetworkOperator().substring(0,3)	310	is emulator or a USA device (MCC)‡
TelephonyManager.getNetworkOperator().substring(3)	260	is emulator or a T-Mobile USA device (MNC)
TelephonyManager.getPhoneType()	1	possibly emulator (GSM)
TelephonyManager.getSimCountryIso()	us	possibly emulator
TelephonyManager.getSimSerial Number()	89014103211118510720	is emulator OR a 2.2-based device
TelephonyManager.getSubscriberId()	310260000000000‡‡	is emulator
TelephonyManager.getVoiceMailNumber()	15552175049	is emulator

Table 1: Listing of API methods that can be used for emulator detection. Some values clearly indicate that an emulator is in use, others can be used to contribute to likelihood or in combination with other values for emulator detection. † xx indicates a small range of ports for a particular emulator instance as obtained by the Android Debug Bridge (ADB). Emulator instances begin at 54 and will always be an even number between 54 and 84 (inclusive). ‡ 310 is the MCC code for U.S. but may also be used in Guam. †† The value is a prefix.‡‡ An emulator will be in the form MCC + MNC + 0's, checking for the 0's is likely sufficient.

augmented with other data. If another check establishes that the device is a Verizon device, but the MNC shows T-Mobile, this may indicate a modified emulator that is returning spoofed values.

Table 1 also contains several values from the `android.os-.Build` class which contains information about the current software build. Retail devices will have system properties that detail the actual production build. An emulator will have build properties based on the SDK build process used to create the emulator binary, such as the `Build.FINGERPRINT`. The fingerprints listed in Table 2 clearly show a pattern followed by the SDK build process. Even though the SDK documentation warns "Do not attempt to parse this value," testing for the presence of "generic," "sdk," or "test-keys" yields perfect results for emulator detection when compared to our sample of physical devices.

Experiments with physical devices led to some counter-intuitive findings for some values. For example, the `Build.ABI` value on the emulator is "armeabi" which is a plausible value for all devices with an ARM processor (nearly all devices). However, the API returned an empty string when used on a Motorola Droid. Similarly, a `Build.HOST` value starting with "android-test" was also found on the Droid. As shown in Table 3, the `Build.HOST` value is not as useful for emulator detection as other `Build` values.

Detecting emulated networking.

The emulated networking environment is often quite different than that found on physical devices. Each emulator instance is isolated from the host PC's network(s) via software. The network address space is always 10.0.2/24. Furthermore, the last octet of the virtual router, host loopback, up to four DNS resolvers, and the

Device	Build.HOST
Emulator	apa27.mtv.corp.google.com
Emulator	android-test-15.mtv.corp.google.com
Emulator	android-test-13.mtv.corp.google.com
Emulator	android-test-25.mtv.corp.google.com
Emulator	android-test-26.mtv.corp.google.com
Emulator	vpbs30.mtv.corp.google.com
Emulator	vpak21.mtv.corp.google.com
Motorola Droid	android-test-10.mtv.corp.google.com
HTC EVO 4G	AA137
Samsung Charge	SEI-26
Samsung Galaxy Tab7	SEP-40
Samsung Galaxy Nexus	vpak26.mtv.corp.google.com

Table 3: Build values collected from various instances.

emulator's address are always known (1, 2, 3–6, and 15, respectively). Unlike ADB, which reserves adjacent, incrementing TCP ports, the network schema is the same for every emulator instance, even if several instances are simultaneously running on one host.

While it is possible that a network to which a real device is connected may exhibit exactly the same network layout, it is unlikely. Devices configured with cellular data plans will often user carrier DNS resolvers and have a carrier DHCP lease for the device IP. WIFI networks to which a device connects are also unlikely to be configured in exactly this way, and are unlikely to exhibit the quiet nature of a /24 network solely allocated to one emulated device.

Accessing network information is relatively straightforward using standard Java techniques. A trivial detection for the emulated

Device	Fingerprint
Emulator	generic/sdk/generic/:1.5/CUPCAKE/150240:eng/test-keys
Emulator	generic/sdk/generic/:1.6/Donut/20842:eng/test-keys
Emulator	generic/sdk/generic/:2.1-update1/ECLAIR/35983:eng/test-keys
Emulator	generic/sdk/generic/:2.2/FRF91/43546:eng/test-keys
Emulator	generic/sdk/generic:2.3.3/GRI34/101070:eng/test-keys
Emulator	generic/sdk/generic:4.1.2/MASTER/495790:eng/test-keys
Emulator	generic/sdk/generic:4.2/JB_MR1/526865:eng/test-keys
Motorola Droid	verizon/voles/sholes/sholes:2.0.1/ESD56/20996:user/release-keys
Motorola Droid	verizon/voles/sholes/sholes:2.2.1/FRG83D/75603:user/release-keys
HTC EVO 4G	sprint/htc_supersonic/supersonic:2.3.3/GRI40/133994.1:user/release-keys
Samsung Charge	verizon/SCH-I510/SCH-I510:2.3.6/GINGERBREAD/EP4:user/release-keys
Samsung Galaxy Nexus	google/mysid/toro:4.1.1/JRO03O/424425:user/release-keys

Table 2: Listing of `Build.FINGERPRINT`'s collected from various instances. Emulator instances clearly include common substrings not found in physical devices.

network would be to check for the four known IP addresses. The false positives on this approach would be low and suffice in most situations. In order to obtain networking information and to interact with the network, the Android application would need to request the ACCESS_NETWORK_STATE and INTERNET permissions.

Detecting the underlying emulator.

The underlying emulator, QEMU, is employed to drive the hardware emulation of the Android emulator. As such, any QEMU detection techniques such as using CPU bugs [33] or virtual address allocation [17] can also be employed against the Android emulator. These techniques, however, require to run so-called native code, that is, software that executes on the processor instead of Android's Dalvik VM. Such software must be compiled with the Native Development Kit (NDK), and is only found in 4.52% of applications [41]. Hence, malware that attempts to detect the underlying emulator using native code may actually be easy to flag as suspicious.

Some CPU based detection mechanisms rely upon privileged instructions which cannot be used as part of a typical application. Android uses Linux for privilege separation and each application is installed under a different user ID. As is common in modern operating systems the application code (both Dalvik and native code) executes in user-mode. In order to use privileged instructions the application must gain elevated privileges by "rooting" the device. While this is certainly possible [37], many of the other techniques discussed in this paper appear much simpler to deploy.

3.2 Differences in performance

The emulator, unassisted by hardware, is at a natural disadvantage when it comes to processing speed. Translating the instructions for execution indeed inherently causes a performance penalty. However, the emulator is typically run on a PC with considerably more processing resources than a mobile device. Thus, it is plausible that, even with the instruction translation penalty, an emulator could approximate the performance of a physical device. We tested this hypothesis by 1) measuring CPU performance and 2) measuring graphical performance.

3.2.1 CPU performance

Some hardware features often used for performance measurements are unavailable to the application executing in user-mode. For example, the performance counters on ARM processors are enabled via the CP15 register and may only be enabled from the privileged kernel—which will not be accessible unless the phone is rooted. This limitation also poses a significant barrier to using many processor benchmarking software suites, which require higher privilege than an Android application possesses.

Device	Average Round Duration (Seconds)	Standard Deviation
PC (Linux)	0.153	0.012
Galaxy Nexus (4.2.2)	16.798	0.419
Samsung Charge (2.3.6)	22.647	0.398
Motorola Droid (2.2)	24.420	0.413
Emulator 2.2	62.184	7.549
Emulator 4.2.2	68.872	0.804

Table 4: Pi calculation round duration on tested devices using AGM technique (16 rounds). The tested devices are noticably slower at performing the calculations than related devices running similar software.

We instead turn to a crude method of benchmarking: evaluating the duration of lengthy computations. We created a Java Native Interface (JNI) application for Android using the NDK. Our application calculates the first 1,048,576 digits of Pi using the method described by Ooura [29]. Pi is calculated over 16 rounds of increasing precision using Arithmetic-Geometric Mean (AGM) making extensive use of Fast Fourier Transforms, square roots, multiplication, etc.

The AGM calculation shows significantly different results when comparing emulated instances with similar physical devices. Table 4 shows the average and standard deviation for several tested devices. For instance, executing the native application on a 4.2.2 emulator results in an median round duration of 68.8 seconds with a total execution time of 1101.9 seconds. A 4.2.2 Galaxy Nexus demonstrates a 16.8 second median round duration, and takes 268.8 seconds to complete. Comparatively, the PC hosting the emulator executes the same code 2.5 seconds with a median of 0.15 second/round (Linux PC). Round durations show statistically significant differences between emulated and non-emulated environments (Wilcoxon test, $W = 256$, $p < .001$), which in turn demonstrates the viability of the this emulation detection technique.

3.2.2 Graphical performance

We also investigated differences in the video frame rate across various hardware and emulator instances. To that effect, we created an Android application that uses Android's `SurfaceHolder` callback method as part of the `SurfaceView` class. A thread continuously monitors the `SurfaceHolder` canvas and calculates a

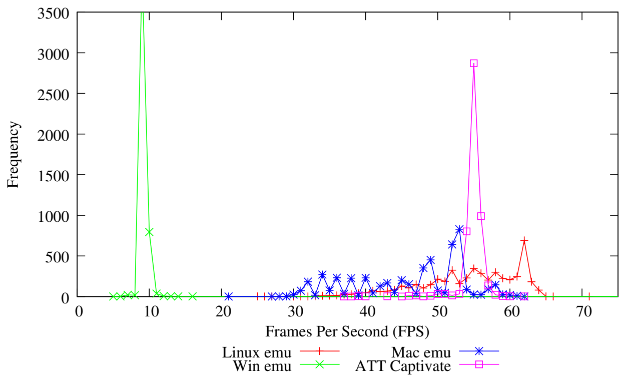

Figure 1: Android 2.2 FPS measurements: Windows exhibits a very low frame rate. The hardware device, ATT captivate, clearly has a very tight bound on framerate at 58-59 FPS. All emulators are statistically different from the real device (Wilcoxon, $p < .001$).

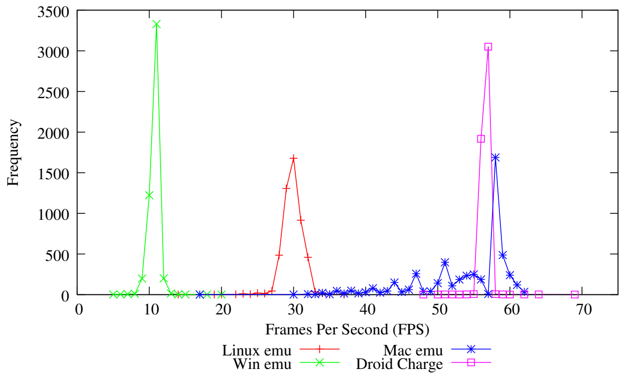

Figure 2: Android 2.3.x FPS Measurements: The Droid Charge registers the majority frame readings between 57 and 58, while the emulators show lower, more distributed readings. The emulators are 2.3.3; the Droid Charge is 2.3.6. All emulators are statistically different from the real device (Wilcoxon, $p < .001$)

Frame Per Second (FPS) value. We sampled 5,000 values for each device.

We ran each emulator instance serially on a 4-core i7 host with 8 GB of RAM and at no time did the the system monitor indicate that any PC resources were a limiting factor. Each emulator and physical device was cold booted and left idle for five minutes prior to testing, and the instances were not otherwise used.

Figures 1–3 show that physical devices typically exhibit both a higher and much more consistent FPS rate. For example, the Galaxy Nexus (4.2.2) shows 58 or 59 FPS for quite near the duration of the experiment. However, a similar emulator shows both much lower FPS near 11 and exhibits more of a bell curve with observable measurements between 9 and 14. Also, some frame rates simply would not occur on a commodity device, such as the 150+ FPS reported occasionally by a 3.2 emulator.

From the experimental data, heuristics can be devised to detect emulators in a host-OS agnostic way across all versions of Android. One such heuristic is shown in Figure 4. The intuition behind the heuristic is that the emulators generally exhibit slower and less tightly bounded FPS rates, often representing a bell curve. Each of the physical devices, on the other hand, exhibit relatively high

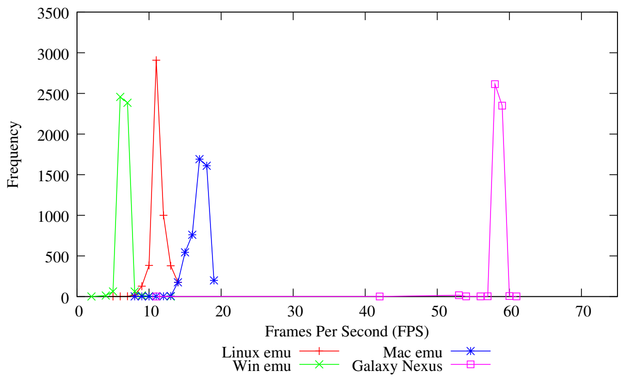

Figure 3: Android 4.2.2 FPS Measurements: Emulators clearly show a low rate, and more of a bell curve than the Galaxy Nexus which shows almost entirely 59-60 FPS. All emulators are statistically different from the real device (Wilcoxon, $p < .001$).

```
for a value: v
 IF
  > 80% of samples fall within v-1..v+1
 AND
  v is > 30 FPS
 THEN
  the sampled device is a physical device
 ELSE
  the sampled device is an emulator
```

Figure 4: Sample heuristic using Frames Per Second (FPS) to determine if an Android application is executing on a physical device, indicated be high and closely-coupled FPS values.

and tightly coupled rates. By identifying FPS distributions that are less coupled or slower, emulator instances can be identified.

Versions of the Android SDK after Revision 17 allow for limited emulator GPU acceleration, though only on Mac and Windows systems. This feature is only meant to work for emulator instances of Android 4.0.3 or later [11], but we tested several configurations anyway. Figures 5 and 6 show FPS results from Mac and Windows emulators and GPU-assisted emulators.

On the Mac, the 4.2.2 emulator instance, the only supported platform, appears to behave considerably more like a physical device than the 4.2.2 emulator without assistance from the host GPU. However, the GPU-assisted 4.2.2 emulator still registers visible FPS rates in the 30-60 and 60-65 ranges, not the tightly coupled plot of almost exclusively 59-60 FPS as observed in Figure 3. The difference between the GPU-assisted 4.2.2 emulator on the Mac and a real Galaxy Nexus, like all of the timing results, is statistically significant (Wilcoxon test, $W = 8.1 \times 10^6$, $p < .001$). On the other hand, GPU assistance on Windows emulators does not considerably improve upon collected values.

3.3 Differences in components

Modern devices are composed of complex combinations of hardware subsystems that are all meant to work in concert. These subsystems all have their own physical implementations and often vary from what is provide with the Android emulator. Likewise, devices are shipped with proprietary software that drives special hardware, interacts with web services, or implements specific functions such as digital rights management. These hardware and software components can both be used to differentiate physical devices from virtual instances.

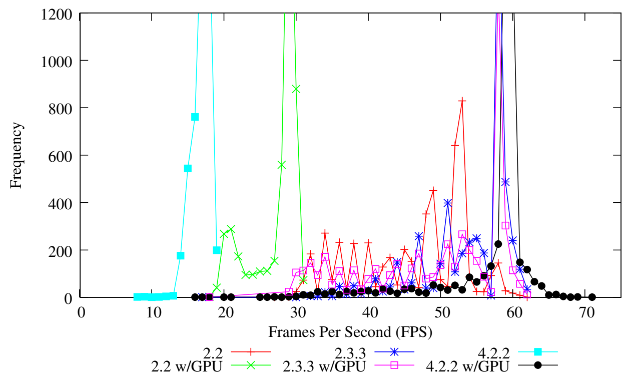

Figure 5: GPU-Accelerated Emulator (Mac): The only supported version, 4.2.2, shows significant improvement when using the host GPU (AMD Radeon 6750M - 1GB RAM). The GPU-assisted 4.2.2 emulator more closely resembles a physical device, but still exhibits a visible tail through the 30-60 FPS range.

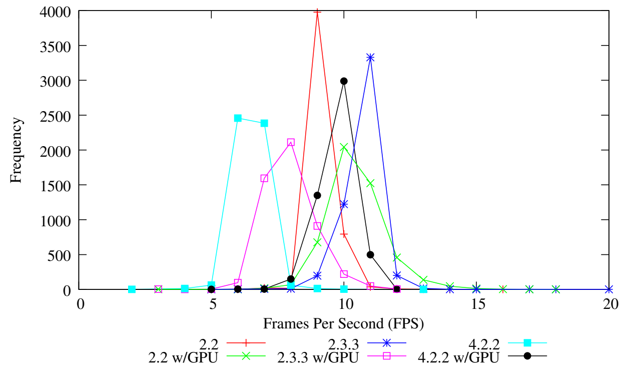

Figure 6: GPU-Accelerated Emulator (Windows): Using the GPU (NVidia 545 GT - 1GB RAM) in Windows did not have a significant effect Note: the scale of this graph is much different than other FPS graphs in this section.

3.3.1 *Hardware components*

With errors in hardware design, such as CPU bugs, indistinguishable emulation of a processor is an arduous task. Emulation of a complete hardware system is even more difficult. Since emulated environments must appear similar to a physical device, other components such as I/O ports, memory management chips and networking devices must all somehow be made available to emulated software. Similar to virtual IDE/SCSI devices exhibiting certain characteristics that facilitate PC emulator detection [33], differences can be observed in the Android emulator. We focus on two classes of differences, those observable due to emulated hardware (or lack of) and those observable due to omitted software.

Hardware components.

Much like specific hardware values present in PC components, values for various hardware components are observable in Android. For example, the CPU serial number is world-readable as part of `/proc/cpuinfo`. Emulator CPUs always show a value of sixteen 0's, while real ARM CPUs return a unique 16-character hexadecimal string. Similarly, current CPU frequencies can be retrieved from `/sys/devices/system/cpu-/cpu0/cpufreq/cpuinfo_min_f req` and `max_freq` on a Galaxy Nexus, but these files are not present in a 4.2.2 emulator.

Sensor	**Android Version**	**Moto. Droid**	**Samsung Charge**	**HTC EVO 4G**	**Galaxy Nexus**
Accelerometer	1.5	1	1	1	1
Ambient Temperature	4.0	-	-	-	0
Gravity	2.3	-	1	1	2
Gyroscope	1.5	0	1	0	2
Light	1.5	1	1	1	1
Linear Acceleration	2.3	-	1	1	2
Magnetic Field	1.5	1	1	1	1
Orientation	1.5	1	1	1	2
Pressure	1.5	0	0	0	1
Proximity	1.5	1	1	1	1
Relative Humidity	4.0	-	-	-	0
Rotation Vector	2.3	-	1	1	2
Temperature	1.5	1	0	0	0
Total		6	9	8	15

Table 5: `Sensor` types, the earliest version of Android to support each type, and observed sensor counts on four devices.

In addition to board-level design decisions such as the use of different memory types [38], devices employ a myriad of motion, environmental and positional hardware sensors often not found on PCs. Even budget devices often have GPS receivers, Bluetooth, accelerometers, temperature sensors, ambient light sensors, gravity sensors, etc. More recent or expensive devices often have additional capabilities perceived as market-differentiators such as Near Field Communication (NFC) chips, air pressure sensors, or humidity sensors.

The sensor types supported as of API 14 (Android 4.x) are shown in Table 5. Some types of sensors are not natively supported on older devices. Observing the type and quantity of sensors on a particular device can easily be performed via an the `Sensor-Manager` API. The size of the list returned from `getSensor-List()` for each type of sensor shown in Table 5 reveals the quantity of each type. Even early devices such as the Motorola Droid have many types of sensors.

Simply observing the number of devices may be sufficient for emulator detection, but this metric is relatively easy to spoof by modifying the SDK. A modified emulator may simply return lists of an appropriate size for each sensor type. More advanced emulator detection could be performed by interacting with each sensor. This type of emulator detection would require significant modification to the emulator, such as emulating the entire functionality of a sensor along with synthetic data.

Recent versions of the SDK facilitate adding some virtual hardware to an emulator instance. Figure 7 enumerates the configuration settings for several supported device types, one of each may be added to an emulator instance. However, this support is limited to later versions of Android, and the degree to which the virtual device emulates a physical device varies.

As an example, we implemented an application to continuously monitor the accelerometer, a very common sensor that requires no permission to be accessed. Since we are interested gathering data as close as possible to the real-time readings we poll using the fastest setting (`SENSOR_DELAY_FASTEST`). Since the accelerometer measures acceleration (m/s^2) along all three axes, we subtract the acceleration force of gravity (9.81) from the vertical

```
hw.sensors.temperature=yes
hw.camera.back=emulated
hw.gpu.enabled=yes
hw.gsmModem=yes
hw.sensors.magnetic_field=yes
hw.accelerometer=yes
hw.audioOutput=yes
hw.battery=yes
hw.sensors.proximity=yes
hw.lcd.backlight=yes
hw.sensors.orientation=yes
hw.camera.front=emulated
hw.gps=yes
```

Figure 7: Android Virtual Device (AVD) configuration settings to add emulated hardware. These settings can be added to the device ini file in order to indicate the virtual presence of hardware sensors such as the accelerometer, or temperature sensor.

axis to normalize the recorded values. In this way, an accelerometer of a stationary device should register 0 for all three axes.

Figure 9 shows measurements for a physical device are closely coupled, but are neither always constant for any axis nor 0. A virtual device with an emulated accelerometer yields exactly the same value on every axis for every sampled data point. Regardless of Android version in the emulator or the host OS on which the emulator is used, the values are always 0.0, 9.77622, 0.813417 (x, y, and z). A device that is actually being used will show a drastically wider distribution along all axes.

Similar detections can be created for other sensors, either to detect an emulator exactly, via known emulator values, or by determining a heuristic for detect a physical device based on hardware ranges. Furthermore, similar detections can be created for other hardware subsystems such as Bluetooth which is often found on physical devices but is not present at all in the emulator. Simply testing to see if a valid handle is returned from `BluetoothAdapter.getDefaultAdapter()` will indicate that the device has a Bluetooth capability and is thus a physical device.

Software components relating to hardware.

Detection techniques very similar to the sensor detections discussed above can be created for the camera(s) and for readings akin to sensors such as the battery level. The battery could be monitored over time to ensure the battery level changes or depletes. The exception, of course, is if the device is plugged in and is thus constantly at 100%. The level and charging status can be observed using the `BatteryManager` API as shown in Figure 8. The battery level on the emulator is exclusively set at 50% or the two components are two known constants (level is 0 and scale is 100).

Another detection relates to special software added by manufactures in order to support certain hardware. Manufacturers often add special hardware to device as a market-differentiator to consumers or in order to take advantage of hardware not natively supported by Android. Such support is added via kernel modules, and like many Linux-based systems, Android detections could consist of looking at what might be loaded from `/sys/module` or the `/lib/modules` directory and what is currently inserted via the `lsmod` command or `/proc/modules` interface. As with other detection techniques, kernel module detection can take a high-level approach by counting (a 4.2.2 emulator shows 26, a physical Galaxy Nexus shows 72), or a more granular approach.

One specific example of such software is kernel modules added by Samsung in order to take advantage of Samsung memory and the proprietary RFS (Robust FAT File System) [38] instead of

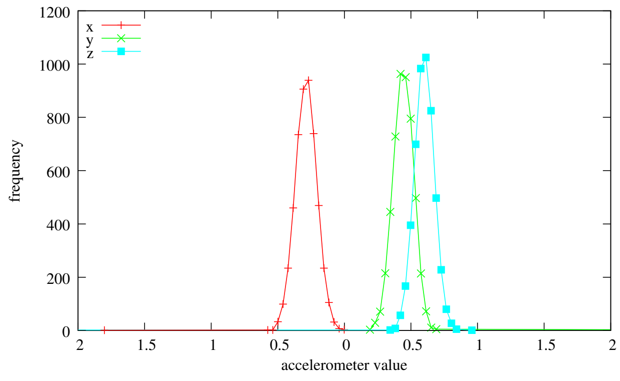

Figure 9: Accelerometer values: Measurements from 5000 data points gathered as quickly as possible from a Samsung Galaxy Nexus (4.2.2) (vertical value adjusted for gravity).

the common Linux Memory Technology Device (MTD) system. Simply listing `/proc/modules` on the Samsung Charge reveals that modules for RFS are loaded on the device. To improve detection, the compiled modules (`/lib/modules/rfs_fat.ko` and `rfs_glue.ko`) can be inspected for `modinfo`-like information or for specifically known byte sequences.

3.3.2 Software components

In addition to the software that specifically supports hardware, consumer Android devices are shipped with a variety of additional software. This software ranges from pre-installed applications to services designed specifically to interface with "cloud" resources or enable Digital Rights Management (DRM). Clear omissions from the emulator are the applications and supporting software for Google's Internet services. The marketplace application, Google Play, the Google Maps API, Google Talk, the Google services used to handle authentication and session information, and other similar software are found on nearly every consumer Android device, but are not included in the emulator. Observing `GoogleLoginService.apk`, `GoogleServicesFramework.apk`, `Phonesky.apk` or `Vending.apk` in `/system/app` likely indicates a real device. Carrier-added applications such as Verizon's backup software (`VZWBackupAssistant.apk`) can be observed the same way and similarly indicate real device.

Instead of or in addition to inspecting files, an API can be used to query the installed applications in an instance. The `PackageManager`'s `getInstalledPackages(0)` interface can be used to obtain a list of all installed applications. The list can then be queried for the Java-style package names such as `com.google.android.gsf` or `com.android.vending`. The associated application for each list item can also be located indirectly through `applicationInfo.sourceDir` which will provide the full path to the APK.

Android's `ContentResolver` can also be used to query the system for the presence of a software service. For instance, the Google Services Framework identifier can be queried with `ContentResolver.query(content: //com.google.android.gsf.services, null,null,"android_id",null)`. The emulator does not support this service and will always return `null` for this query.

In addition to observing the presence (or absence) of software services, variations in software component behavior can be observed. For instance, when establishing an interactive shell to an instance via ADB, the behavior is different between an emulator

```
int level = batteryStatus.getIntExtra(BatteryManager.EXTRA_LEVEL, -1);
int scale = batteryStatus.getIntExtra(BatteryManager.EXTRA_SCALE, -1);
float batteryPct = level / (float)scale;

boolean isCharging = status == BatteryManager.BATTERY_STATUS_CHARGING ||
                     status == BatteryManager.BATTERY_STATUS_FULL;
```

Figure 8: Battery level emulator detection example. The battery level is obtained via two Android Intents [8]. If batteryPct is exactly 50% or the level is exactly 0 and the scale is exactly 100, the device in question is likely an emulator. The level could be monitored over time to ensure it varies, and the charging status could be used to determine if the battery should be constant (at 100%).

and a physical device. In particular, in an emulator shell the effective Linux user id is root (0). This difference is caused by a check in ADB called `should_drop_privileges` which inspects two "read-only" system properties: `ro.kernel.qemu` and `ro.secure`. The inspection verifies that the instance is running the emulated kernel and that the "secure" setting is zero meaning that the root shell should be permitted.[1]

There are no APIs for inspecting properties such as `ro.secure`, but the properties are loaded from the `default.prop` file at system boot. Standard Java file methods can be used to read this file and therefor examine settings such as `ro.secure`. Even though this file is not likely to be modified, obtaining the properties in this way may not reflect the runtime state of the instance. We will explore other ways of obtaining actual runtime values in Section 4.

3.4 Differences due to system design

Modern runtime analysis systems must cope with certain constraints that do not affect consumer devices. In particular runtime systems must generally process a considerable volume of malware as the number of daily unique samples is quite large and reportedly growing. This phenomenon has been observed for years in the realm of PC malware and early signs indicate a similar growth pattern for mobile malware. Unfortunately, this requirement for additional scale is often at odds with resource availability. It is simply not economically viable to purchase thousands of physical devices or to run thousands of emulators simultaneously, forever. For these reasons, critical design decisions must be made when designing a runtime analysis system and the effects of these decisions can be used as a form of runtime-analysis detection. We outline two classes of design decisions that may influence an attackers ability to detect or circumvent analysis: those shared with PC runtime analysis systems and those specific to Android.

PC system design decisions, such execution time allotted to each sample or how much storage to allocate to each instance, have been explored in the PC realm. Many of these same decisions must be made for a mobile malware runtime analysis system. System circumvention, such as delaying execution past the maximum allotted execution time, is also shared with PC techniques.

Android-specific design decisions revolve around the inherent differences between a device that is actively used by an individual and a newly created instance (virtual or physical). If an attacker can determine that a device is not actually in use, the attacker may conclude that there is no valuable information to steal or that the device is part of an analysis system.

Metrics for determining if the device is (or has been) in use include quantities such as the number of contacts and installed applications, and usage indicators such as the presence and length of text messaging and call logs. These and many more indicators are available programatically as part of the Android API, but many require the application to request particular permissions. Runtime analysis system detection using these metrics is not as clear-cut as the other techniques we presented. These values depend upon knowing the average or minimum quantities present on the typical consumer device, and would be rife with false positives if the quantities were not evenly distributed among all users. Some work shows that these values are indeed not evenly distributed as a small user study showed eight of 20 participants downloaded ten or fewer applications while two of 20 downloaded more than 100 [23].

4. MINIMIZING THE PERMISSIONS NEEDED

Some of the detections mentioned in this paper require certain application-level permissions. For example, to detect if Bluetooth is present on a device the application must request the `android.permission.BLUETOOTH` or `BLUETOOTH_ADMIN` permission. Other resources, such as the accelerometer require no permission to access.

All the techniques described in previous sections require only a very limited set of application permissions and the make use of existing APIs that are unlikely to change substantially in the foreseeable future. However, using the advertised APIs also has a offensive drawback: Designers of automated analysis systems could attempt to address the specific APIs we have utilized instead of completely addressing the downfalls of the emulation environment. For example, mitigating the device ID check by simply hooking the call to `TelephonyManager.getDeviceId()` and causing the return value to not be all 0's. To demonstrate how such an approach is short-sighted defensively, we present two techniques for retrieving runtime system properties even though there is no programmatic API in the SDK. We can then use these properties to create alternate implementations of nearly every detection we have detailed.

We offer two additional techniques for obtaining system properties: reflection and runtime `exec` (subprocess). Example code can be found in Figure 11 in the Appendix. In the reflection example, the `android.os.SystemProperties` class is obtained via the `ClassLoader`. The `get` method can then be obtained from the class and used to retrieve specific properties. This example retrieves settings we've already discussed, such as the `ro.secure`, the battery level and the `Build` tags. The `exec` example is more general and simply prints a list of all runtime properties be executing the "getprop" binary and parsing the output.

5. EVALUATION

Our techniques were developed in a relatively isolated test environment. We thus need to measure the effectiveness of our techniques against real analysis systems. We identified publicly available Android sandbox systems via literature review, industry referral, and via Internet searches. Candidate systems had to have a public, automated interface to be considered. Ideally, a candi-

[1]The method also employs a second check for `ro.debuggable` and `service.adb.root`; if these are both 1, privileges will not be dropped even if `ro.secure` is set to 1.

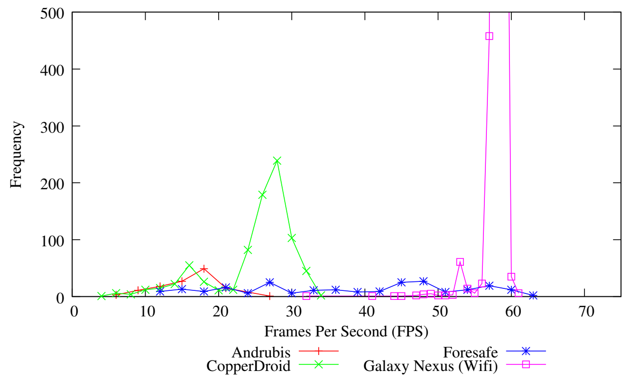

Figure 10: FPS measurements for sandboxes: For comparison, a physical Galaxy Nexus was re-measured using the same application. The physical device shows strong coupling at 59 FPS and all of the sandboxes demonstrate loose coupling and wide distribution, indicating that they all make use of virtualization.

date system also provides a publicly accessible output – typically as some form of report.

Our candidate sandboxes were Andrubis [2], SandDroid [10], Foresafe [6], Copperdroid [3], AMAT [1], mobile-sandbox [7] and Bouncer [25]. Each sandbox presents a slightly different interface, but are all meant to be accessible as web services. Likewise, each sandbox is the result of different levels of developmental effort and therefore embodies various levels of product maturity.

Mobile-sandbox constantly displayed a message indicating that 0 static and 308,260 dynamic jobs were yet to be processed. We were only ever able to observe static analysis output of mobile-sandbox. Similarly, SandDroid seemed to not route or otherwise filtered outbound network traffic, and the SandDroid reports only displayed results of static analysis so it was not possible to test our evasion techniques on SandDroid.

AMAT's online instance does not appear to be in working order, immediately stating that any uploaded application was "not malware." AMAT did not provide any further reasoning as to why this message was displayed, but uploading APK files did result in the overall analysis number incrementing with each submission. When using Foresafe, an action would occasionally fail, and a server-side error would be displayed to the user. Even so, refreshing the page and repeating the action always seemed to solicit the desired effect.

Google's Bouncer does not immediately meet our minimal requirement of having a public interface, but we attempted to include it given its importance on deployed applications. Not much about the inner workings of Bouncer has been made available. Without the ability to directly submit applications for analysis, and without the ability to directly view reports, interaction with Bouncer is widely unpredictable. Indeed, even after submitting several, increasingly offensive, applications into the Google Play marketplace, we never observed any connections coming from Bouncer. It is possible that Bouncer has altered the decision process for selecting which applications to run (e.g. only those with more than 10K downloads, or those with negative reviews) or has been changed to disallow connections to the Internet following other work on profiling the inner workings of Bouncer [28, 31].

5.1 Behavior evaluation

As shown in Table 6, the SDK and `TelephonyManager` detection methods prove successful against all measured sandboxes. Many of the simple detection heuristics outlined in Table 1 are similarly successful. However, some of the `Build` parameters, such as `HOST`, `ID`, and `manufacturer` require a more complex heuristic in order to be useful. Detecting the emulated networking environment was also very successful as the sandboxes all employed the default network configuration.

Detection method	**Andrubis**	**CopperDroid**	**ForeSafe**
getDeviceId()	Y†	Y	Y
getSimSerial Number()	Y	Y	Y
getLine1 Number()	Y	Y‡	Y
MCC	Y	Y	Y
MNC	Y	Y	Y
FINGERPRINT	Y	Y	Y
BOARD	Y	Y	Y
BRAND	Y	Y	Y
DEVICE	Y	Y	Y
HOST	N	N	N
ID	N	N	N
manufacturer	N	N	N
MODEL	N	N	Y
PRODUCT	N	N	Y
serial	Y	N	N
TAGS	Y	Y	Y
radio	N	N	N
USER	N	N	N
NetCountry	y	N	N
NetType	y	N	N
PhoneType	y	N	N
SimCountry	y	N	N
VMNum	Y	Y	Y
SubscriberID	Y†	Y	Y
Networking	Y	Y	Y
bluetooth	Y	Y	Y
ro.secure	Y	Y	Y
sensors	Y	Y	Y
contacts	Y	Y	Y
call log	Y	Y	Y
performance	Y	Y	Y

Table 6: Evaluation of detections. An uppercase Y indicates that the system was detected as an emulator, a lowercase y indicates that the system may be an emulator, and an uppercase N indicates that the detection mechanism did not succeed. † This detection was actually designed to detect a particular tool in the same manner as described in Section 3.1, and we discuss the detection in the Section 6.‡ the number is not exactly an emulator number, but the area code is 555 which is not valid.

5.2 Performance evaluation

The graphical performance measurements further indicate that all of the measured sandboxes are built using virtualization. Figure 10 shows measurements from the sandboxes as well as a hardware device. As with the emulators sampled in section 3.2, each of the sandboxes exhibit a lower, loosely coupled values. Unlike in our own test environment, we have no control over the duration of execution in the measured sandboxes. Due to premature termination, we only received a subset of the 5,000 measurements the application should have generated (604, 814, and 229 for Andrubis, CopperDroid, and Foresafe, respectively).

Sensor	Andrubis	CopperDroid	ForeSafe
Accelerometer	0	1	0
Ambient Temperature	0	0	0
Gravity	0	0	0
Gyroscope	0	0	0
Light	0	0	0
Linear Acceleration	0	0	0
Magnetic Field	0	0	0
Orientation	0	0	0
Pressure	0	0	0
Proximity	0	0	0
Relative Humidity	0	0	0
Rotation Vector	0	0	0
Temperature	0	0	0
Total	0	1	0

Table 7: `Sensor` counts, as evaluated from different Android sandbox systems. Very different from table 5, the sandboxes exhibit very few sensors.

5.3 Hardware and software component evaluation

None of the measured sandbox systems made effort to emulate hardware sensors. As shown in Table 7, only CopperDroid handles accelerometer events, and no sandbox handles any other sensor type. Likewise, the sandboxes report exact, unchanging values for the battery level (0) and scale (100) as used in the emulator.

5.4 System design evaluation

We elected to only measure coarse data in regard to potentially sensitive items such as address book contacts and call logs. With the wide distribution of our test applications to public sandboxes and Google Play, we wanted to reduce the risk of incidentally receiving any private information. For this reason, we only collected the total counts of contacts, call logs, and applications installed. Even so, these total counts are likely enough to discern if a device is realistically in use as the sandboxes report very low values. Better heuristics could certainly be developed over time. Andrubis reports 63 applications installed and Foresafe reports 46 apps, but neither sandbox has any of the Google services apps such as Google Talk, YouTube, or Google Service Framework installed. Andrubis shows 5 contacts and 3 call logs, indicating that some thought has been given to making Andrubis look like a device in use. The standard emulator has no contacts and no call logs, which is also what Foresafe shows.

In addition to the duration shown in the various sandbox reports, we attempted to measure the execution time. We did this by creating an application that embedded the system timestamp (`System.currentTimeMillis()`) in repeated UDP packets sent out from the sandbox. By comparing the largest and smallest value received, we can approximate the duration of execution. For instance, Foresafe may report a "Processing Time" of 106 seconds, yet for the same submission we observe timestamp values indicating an execution time of 68 seconds. Similarly, When Andrubis reports an analysis duration of 306 seconds we observe 248 seconds and when CopperDroid reports 731 seconds, we observe 399 seconds. The additional time listed by each sandbox may include some operations other than exclusively executing the application.

6. DISCUSSION

Some of the emulation detections presented in this work are relatively simple to mitigate. For instance, unexpected values could be returned for many of the API methods listed in Table 1. Some security tools, such as DroidBox [4], have started to take this step. However, DroidBox does not raise the bar very high as the new values are set to a known constant. For example, the `TelephonyManager` device identifier is always 357242043237517, and the subscriber identifier is always 310005123456789 [5]. When other security tools make use of software packages such as DroidBox, they become subject to evasion due to DroidBox detections. Such is the case for Andrubis. A sandbox system designer must be thoughtful about *how* these detections are mitigated. As demonstrated in section 4, reflection and runtime exec can be used to observe many system values in different ways, a good mitigation will handle all such cases.

Perhaps the easiest mitigations to counter are those that rely on the high-level state of the device, such as the number of contacts and the call log. Populating a sandbox with a copious address book makes such a detection considerably more difficult for an attacker.

Yet other detection mechanisms we have presented are far more difficult to counter. Techniques rooted in detecting virtualization itself, such as those we presented via timing, present a difficult hurdle for mobile sandbox system designers as they essentially require to redesign the emulator to obtain timing measurements that closely resemble those obtained on actual hardware. While it may be possible to reduce the wide discrepancy we have observed through our measurements, one can easily imagine the next step of this arms race would be to build up a hardware profile based on various measurements (CPU, graphics, ...) over several benchmarks rather than the simple Pi calculation we relied upon. We conjecture it could be possible to pinpoint the exact hardware used with such a technique—and of course, to detect any emulation.

7. CONCLUSION

As with many malware-related technologies, the detection of dynamic analysis systems is one side of an arms race. The primary reason emulator detection is more applicable here than for PCs is that practical use cases have developed for virtualizing general purpose computers – a phenomenon that has yet to occur for mobile devices. Virtualization is not broadly available on consumer mobile platforms. For this reason, we believe that mobile-oriented detection techniques will have more longevity than corresponding techniques on the PC.

We have presented a number of emulator and dynamic analysis detection methods for Android devices. Our detections are rooted in observed differences in hardware, software and device usage. From an implementation perspective, the detection techniques require little or no access beyond what a typical application would normally be granted. Such detections can significantly raise the bar for designers of dynamic analysis systems as they must universally mitigate all detections. Of those, hardware differences appear to be the most vexing to address: a very simple evaluation of the frame-per-second rate immediately led us to identify malware sandboxes, without requiring advanced permissions. Likewise, accelerometer values would yield definitive clues that the malware is running in a sandboxed environment. Whether concealing such hardware properties can be done in practice remains an open problem, on which we hope the present paper will foster research, lest malware sandboxes be easily detected and evaded.

8. REFERENCES

[1] AMAT: Android Malware Analysis Toolkit. `http://sourceforge.net/projects/amatlinux/`.

[2] Andrubis. `http://anubis.iseclab.org/`.

[3] CopperDroid. `http://copperdroid.isg.rhul.ac.uk/copperdroid/`.

[4] DroidBox. `https://code.google.com/p/droidbox/`.

[5] Droidbox device identifier patch. `https://code.google.com/p/droidbox/source/browse/trunk/droidbox23/framework_base.patch?r=82`.

[6] Foresafe. `http://www.foresafe.com/scan`.

[7] mobile-sandbox. `http://mobilesandbox.org/`.

[8] Monitoring the Battery Level and Charging State | Android Developers. `http://developer.android.com/training/monitoring-device-state/battery-monitoring.html`.

[9] North American Numbering Plan Adminstration search. `www.nanpa.com/enas/area_code_query.do`.

[10] SandDroid. `http://sanddroid.xjtu.edu.cn/`.

[11] Using the Android Emulator | Android Developers. `http://developer.android.com/tools/devices/emulator.html`.

[12] U. Bayer, P. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *NDSS*, 2009.

[13] T. Blasing, L. Batyuk, A. Schmidt, S. Camtepe, and S. Albayrak. An android application sandbox system for suspicious software detection. In *MALWARE'10*, 2010.

[14] D. J. Chaboya, R. A. Raines, R. O. Baldwin, and B. E. Mullins. Network intrusion detection: automated and manual methods prone to attack and evasion. *Security & Privacy, IEEE*, 4(6):36–43, 2006.

[15] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Dependable Systems and Networks With FTCS and DCC, 2008. IEEE International Conference on*, pages 177–186, 2008.

[16] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Operational experiences with high-volume network intrusion detection. In *Proc. CCS*, pages 2–11. ACM, 2004.

[17] M. F. and P. Schulz. Detecting android sandboxes, Aug 2012. `https://www.dexlabs.org/blog/btdetect`.

[18] A. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *Proc. SPSM*, pages 3–14. ACM, 2011.

[19] P. Ferrie. Attacks on more virtual machine emulators. *Symantec Technology Exchange*, 2007.

[20] P. Fogla and W. Lee. Evading network anomaly detection systems: formal reasoning and practical techniques. In *Proc. CCS*, pages 59–68. ACM, 2006.

[21] M. Handley, V. Paxson, and C. Kreibich. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proc. USENIX Security*, 2001.

[22] T. Holz and F. Raynal. Detecting honeypots and other suspicious environments. In *Information Assurance Workshop, 2005. IAW'05. Proceedings from the Sixth Annual IEEE SMC*, pages 29–36. IEEE, 2005.

[23] P. G. Kelley, S. Consolvo, L. F. Cranor, J. Jung, N. Sadeh, and D. Wetherall. A conundrum of permissions: Installing applications on an android smartphone. In *USEC'12*, pages 68–79. Springer, 2012.

[24] B. Lau and V. Svajcer. Measuring virtual machine detection in malware using dsd tracer. *Journal in Computer Virology*, 6(3):181–195, 2010.

[25] H. Lockheimer. Android and Security, Feb 2012. `http://googlemobile.blogspot.com/2012/02/android-and-security.html`.

[26] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, 2007.

[27] D. Mutz, G. Vigna, and R. Kemmerer. An experience developing an ids stimulator for the black-box testing of network intrusion detection systems. In *Computer Security Applications Conference, 2003. Proceedings. 19th Annual*, pages 374–383. IEEE, 2003.

[28] J. Oberheide and C. Miller. Dissecting the android bouncer. *SummerCon2012, New York*, 2012.

[29] T. Ooura. Improvement of the pi calculation algorithm and implementation of fast multiple precision computation. *Transactions-Japan Society for Industrial and Applied Mathematics*, 9(4):165–172, 1999.

[30] 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 *Proc. WOOT*, volume 41, page 86. USENIX, 2009.

[31] N. J. Percoco and S. Schulte. Adventures in bouncerland. *Black Hat USA*, 2012.

[32] T. H. Ptacek and T. N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, DTIC Document, 1998.

[33] T. Raffetseder, C. Krügel, and E. Kirda. Detecting system emulators. In *Information Security*. Springer, 2007.

[34] J. Rutkowska. Red pill... or how to detect vmm using (almost) one cpu instruction. *Invisible Things*, 2004.

[35] T. Strazzere. Dex education 201 anti-emulation, Sept 2013. `http://hitcon.org/2013/download/Tim\%20Strazzere\%20-\%20DexEducation.pdf`.

[36] T. Vidas and N. Christin. Sweetening android lemon markets: measuring and combating malware in application marketplaces. In *Proc. 3rd CODASPY*, pages 197–208. ACM, 2013.

[37] T. Vidas, D. Votipka, and N. Christin. All your droid are belong to us: A survey of current android attacks. In *Proc. WOOT*. USENIX, 2011.

[38] T. Vidas, C. Zhang, and N. Christin. Toward a general collection methodology for android devices. *DFRWS'11*, 2011.

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

[40] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proc. IEEE Symp. on Security and Privacy*, 2012.

[41] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proc. NDSS*, 2012.

9. APPENDIX

```
private void logRuntimeSystemPropsReflect(){
  logAprop("ro.secure");
  logAprop("ro.product.name");
  logAprop("ro.debuggable");
  logAprop("status.battery.level_raw");
  logAprop("ro.build.host");
  logAprop("ro.build.tags");
  logAprop("net.gprs.local-ip");
  logAprop("net.eth0.gw");
  logAprop("net.dns1");
  logAprop("gsm.operator.numeric");
  logAprop("ro.kernel.qemu");
  logAprop("ro.kernel.qemu.gles");
  logAprop("ro.kernel.android.qemud");
}
private void logAprop(String s){
  Log.e("reflect."+s,getPropViaReflect(s));
}

//the actual reflection to obtain a handle to the hidden android.os.SystemProperties
private String getPropViaReflect(String s){
  String ret="";
  try{
    ClassLoader cl = theActivity.getBaseContext().getClassLoader();
    Class<?> SystemProperties = cl.loadClass("android.os.SystemProperties");

    @SuppressWarnings("rawtypes")
    Class[] paramTypes = { String.class };
    Method get = SystemProperties.getMethod("get", paramTypes);

    Object[] params = { s };
    ret = (String) get.invoke(SystemProperties, params);
  } catch(Exception e){
    e.printStackTrace();
  }
  return ret;
}

private void logRuntimeSystemPropsExec(){
  try
  {
    String line;
    java.lang.Process p = Runtime.getRuntime().exec("getprop");
    BufferedReader input = new BufferedReader(new InputStreamReader(p.getInputStream()));
    while ((line = input.readLine()) != null)
    {
      //quick line parsing
      int split = line.indexOf("]: [");
      String k = line.substring(1,split);
      String v = line.substring(split+4,line.length()-1);
      Log.e("runprop."+k,v);
    }
    input.close();
  }
  catch (Exception err)
  {
    err.printStackTrace();
  }
}
```

Figure 11: This listing uses reflection (top) and a runtime exec (bottom) to obtain runtime `SystemProperties` information. In either case the information is logged to the system log, for detection purposes this value would be evaluated in accordance with the detection techniques presented in earlier sections. This code listing obtains information in completely different ways than those detailed in the earlier sections of the paper without using the official API and without requiring additional permissions. This code obtains values for the battery level, the Build configuration, network IP settings, and cellular provider MCC and MNC.