BareDroid: Large-Scale Analysis of Android Apps on Real Devices

Simone Mutti [1]
simone.mutti@unibg.it

Yanick Fratantonio [2]
yanick@cs.ucsb.edu

Antonio Bianchi [2]
antoniob@cs.ucsb.edu

Luca Invernizzi [2]
invernizzi@cs.ucsb.edu

Jacopo Corbetta [2]
jacopo@cs.ucsb.edu

Dhilung Kirat [3]
dkirat@us.ibm.com

Christopher Kruegel [2]
chris@cs.ucsb.edu

Giovanni Vigna [2]
vigna@cs.ucsb.edu

[1] Universitá degli Studi di Bergamo
Bergamo, Italy

[2] UC Santa Barbara
Santa Barbara, CA

[3] IBM Research T.J. Watson
Yorktown Heights, NY

ABSTRACT

To protect Android users, researchers have been analyzing unknown, potentially-malicious applications by using systems based on emulators, such as the Google's Bouncer and Andrubis. Emulators are the go-to choice because of their convenience: they can scale horizontally over multiple hosts, and can be reverted to a known, clean state in a matter of seconds. Emulators, however, are fundamentally *different* from real devices, and previous research has shown how it is possible to automatically develop heuristics to identify an emulated environment, ranging from simple flag checks and unrealistic sensor input, to fingerprinting the hypervisor's handling of basic blocks of instructions. Aware of this aspect, malware authors are starting to exploit this fundamental weakness to evade current detection systems. Unfortunately, analyzing apps directly on bare metal at scale has been so far unfeasible, because the time to restore a device to a clean snapshot is prohibitive: with the same budget, one can analyze an order of magnitude less apps on a physical device than on an emulator.

In this paper, we propose BareDroid, a system that makes bare-metal analysis of Android apps feasible by quickly restoring real devices to a clean snapshot. We show how BareDroid is not detected as an emulated analysis environment by emulator-aware malware or by heuristics from prior research, allowing BareDroid to observe more potentially malicious activity generated by apps. Moreover, we provide a cost analysis, which shows that replacing emulators with BareDroid requires a financial investment of less than twice the cost of the servers that would be running the emulators. Finally, we release BareDroid as an open source project, in the hope it can be useful to other researchers to strengthen their analysis systems.

Categories and Subject Descriptors

[**Security and privacy**]: Software and application security

Keywords

Android, Bare-metal Analysis, Evasive Malware

1. INTRODUCTION

To analyze potentially-malicious Android apps at scale, security researchers have developed a variety of virtualized analysis environments, such as Andrubis [34], Google's Bouncer [22], Mobile-Sandbox [30], and JoeSandbox [27]. Virtualized environments are the go-to choice because they are inherently scalable: for example, they can scale horizontally through multi-host parallel execution and, after an app has been analyzed, the analysis environment can be quickly reverted to a trusted state in a matter of seconds before starting the next analysis. This ability of fast restore keeps the analysis overhead to a minimum.

Unfortunately, malware authors are aware of this trend. They are exploiting this trend for evasive purposes by creating malware that collects information about the environment in which it is being executed, and, if an emulator is detected, it suppresses its malicious behavior to evade analysis. This evasive behavior is well-known in the security community [3], and has been observed by researchers in desktop malware [15, 19] as well as in web malware [13, 17].

Researchers have recently evaluated virtualized Android analysis environments [16], and they have found more than 10,000 detection heuristics that an app can use to detect emulated environment. Some issues they have highlighted can be easily mitigated. For example, researchers have found that, to bypass Google's Bouncer, the OBAD [6] malware contains a series of checks, such as `build.MODEL =="google_sdk"`: if these checks trigger, OBAD will not show any malicious behavior. However, there are classes of detection heuristics that do not rely on such simple flag checks; instead, they leverage fundamental differences between real devices and emulators that are challenging, if not impossible, to mitigate. These heuristics include checks on the GPU performance and power consumption profile, and can leverage minute differences such as how the program-counter register is updated in QEMU (this method has been used in [7] to detect Andrubis [34]).

Although Android emulators can be modified to mitigate some of these heuristics and timing attacks, simultaneously mitigating all these detection techniques is extremely cumbersome and challenging. In fact, such modifications would require the hypervisor to behave exactly like a real Android device, and the emulator to realistically simulate a variety of input signals (e.g., the gyroscopic sensors, accelerometers, magnetometers, mobile/WiFi networks). Moreover, it is not clear whether it will ever be possible to make the performance discrepancy undetectable. Note also that this is the attacker's game: a single imperfection in the mimicking of a real device would render the emulator detectable, thus easy to fingerprint and evade.

Despite the significant threat posed by emulator-aware Android malware (as it can evade all current analysis approaches), no solution to this problem has been proposed. In the desktop malware world, a robust approach to evasive malware has been proposed with bare-metal malware analysis [18, 19]: here, Kirat et al. leverage the IPMI remote-administration features (to power-cycle the workers) and iSCSI interface (to attach remote disks to workers) to perform the analysis of evasive malware. Unfortunately, it is not clear how to apply this approach to off-the-shelf Android devices.

In this paper, we present BareDroid, a system that allows for bare-metal malware analysis on off-the-shelf Android devices. BareDroid is designed to scale at a price-point similar to the one offered by emulators. Our evaluation shows that analyzing malware on BareDroid costs, in the worst case, at most twice as much as executing the same volume of suspicious apps inside X86-accelerated emulators. To achieve this, BareDroid focuses on providing a fast way to restore the device to the initial state after each execution, so to minimize overhead.

We also evaluate BareDroid with existing real-world emulator-aware malicious apps and the known detection heuristics that have been developed by the Android security research community [16]. In all cases, our system is effective in eliciting suspicious behaviors that are suppressed when the analysis is performed on an emulator. Note that BareDroid's goal is not to provide a novel malware analysis engine; instead, its goal is to provide a platform on top of which existing and future analysis engines can perform malware detection without the risk of being evaded by the mere presence of an emulator-like environment. This is why, for the benefit of the security community, we release BareDroid as an open source project [21].

The main contributions of this paper are as follows:

- We design and implement BareDroid, a scalable bare-metal malware-analysis platform for Android devices, the first infrastructure of this kind.
- We evaluate BareDroid with emulator-aware malware and the latest emulator detection heuristics, and we show how they are ineffective when run within our infrastructure.
- We evaluate the feasibility of using BareDroid in place of emulator for large-scale analysis of Android apps, and discuss the cost and implementation aspects of this approach.
- We release BareDroid as an open source project.

2. ANDROID EMULATOR DETECTION

In this section we summarize the discrepancies that an Android app can leverage to check whether it is running in an emulator or on a real device. These discrepancies can be divided in three categories: static artifacts, dynamic artifacts, and hypervisor artifacts.

Static artifacts provide an immediate way to differentiate real devices from emulators. These include the flags in Android SDK `Build` class, such as `Build.HARDWARE == "goldfish"`. Host properties can also be leveraged: examples include the mobile device's IMEI [24] (a unique ID for GSM devices), the SIM card's IMSI (a unique ID specific to the SIM card), the number of cores (emulators typically have a single core) [16], available peripherals (e.g., no emulator supports USB on-the go), and the network hardware and its configuration. Static artifacts can typically be removed by developing more complete emulators, so in principle they can be mitigated through engineering effort.

In contrast, dynamic artifacts cannot be removed easily since they are based on the observation that, in an emulator, not all the various interfaces of a real mobile device are fully functional. For example, an app can check whether it can receive SMS from the mobile network. Also, sensor input can be leveraged: current emulators have very limited support for simulating realistic sensor input (e.g., what is the current GPS fix? What GPS satellites are visible?). Moreover, mobile devices contain tens of sensors: they typically have an accelerometer, gyroscope, GPS, barometer, camera, GSM/WiFi/Bluetooth/RFC radios, internal thermometer(s), proximity sensors, magnetometers, and voltmeters (for cores and battery). These sensor readings cannot be trivially replayed, because they can be influenced by the app: for example, making the device vibrate has an effect on the accelerometer, and putting a load on the CPU increases the internal temperature, the voltage/scaling of the cores, and the battery drain.

Finally, the hypervisor itself can be fingerprinted. QEMU, for example, is detectable because of its caching [24] and scheduling [16] policies. Also, the performance of the GPU provides a base for side-channel timing attacks [33].

3. BAREDROID

3.1 Goals and Challenges

Our work aims to design and develop a system, BareDroid, that can be used to perform large-scale analysis of Android apps on real devices. BareDroid should be fast enough that the hardware cost to achieve a given throughput (in terms of number of apps that can be analyzed within a given time slot) is comparable to the cost of doing the same with emulators.

The first challenge consists in developing a *fast restoring* mechanism, so that the *initial state* (known to be working and uncompromised) of a device can be restored with a low overhead. Note that restoring the state of a device is conceptually straightforward. The immediate approach to this consists in restoring each and every partition of the device before each analysis. However, this approach is prohibitively slow (in our experiments, it takes 141 seconds to perform a full restore on a Nexus 5). Also, whereas efficiently restoring the state of an emulator is simple through the *snapshot restore* functionality, it is challenging to achieve the same speed when restoring real devices.

As our system is meant to scale, another challenge consists in scheduling the analysis of many apps on multiple devices and monitoring their hardware status. In emulators, this task is trivial: in fact, once the analysis on a single device is robust, generalizing the analysis to multi-device just requires the development of a software *driver* component to keep track of the devices and send them the apps to be analyzed.

However, in our experience, we found that the robustness of real devices is far from ideal. In particular, hardware components (e.g., devices, USB cables, USB hubs) significantly increase the surface for failure. For example, it is not rare for devices not to boot up properly and for the *driver* to be unable to connect and restart the booting process. As another example, a malfunction of USB hub may not provide enough power, making the connected devices quickly exhaust their battery.

3.2 Threat Model

Our system is designed to allow the analysis of malicious apps. Thus, we assume that an app under analysis might actively attempt to compromise our infrastructure. In our threat model, an app can execute arbitrary code on the device, and it can launch root exploits to perform privilege escalation. The only assumption that our system relies upon is the availability of a kernel-level mechanism to *lock* a partition (i.e., set its permission as read-only). To achieve this, we rely on a SELinux policy (refer to Section 4 for the details): thus, we assume that the SELinux component enforcing this policy cannot be compromised.

We chose this threat model as it provides the best tradeoff between security and performance: in our experience, the behavior of the vast majority of malicious apps is covered by this threat model. That being said, depending on the analysis scenario, malware authors might outgrow this threat model, and compromise the Android kernel. For this reason, throughout this paper we also discuss the scenario in which *no components* can be fully trusted, and we suggest several strategies (see Section 6.2 for more details).

3.3 Approach Overview

To be fast, BareDroid needs to quickly restore a device to its initial uncompromised state. On a device, the data that needs to be restored can be seen as a list of partitions. The most immediate solution to restoring is to simply overwrite the partitions with their original content. However, we have found this to be unacceptably slow; for this reason, in BareDroid we have opted for a significantly faster, albeit more complex, approach.

The various partitions on a device have different roles, and this influences how common it is for a partition to be modified during the analysis of a given Android app. For example, the *system* partitions, $S_1, S_2, \ldots, S_n$, are *usually* not modified, as they contain the bootloader, the code base for the *recovery mode*, and the kernel. On the other hand, the *user* partitions, $U_1, U_2, \ldots, U_n$, are almost always modified during the analysis. In fact, if an app stores even one single file on the file-system, the user partitions will be modified and, therefore, they will need to be restored. Even if a partition is *usually* not modified, an app could try to alter it. For example, a malicious app could first gain root privileges (by launching a root exploit) and then modify system partitions: In this case, BareDroid will restore these partitions.

In general, it is faster to perform a partition integrity check, than to overwrite its content with its clean version. For partitions that are only changed occasionally, the amortized time of performing the integrity check (and a restore only when needed) is lower than the one that would be taken by always restoring the partition. However, if a partition is modified often, it becomes more efficient to always overwrite its content, skipping the integrity check. This last observation motivated us to use two different strategies when ensuring that both system and user partitions are reverted to a clean state before starting the analysis of a given app.

A functionality already present in an unmodified Android device is that the "integrity check" of the system partitions are always performed at boot time: each system partition contains code that, as one of its tasks, performs an integrity check on the content of the *next* partition. That is, partition S_1 checks the integrity of S_2, S_2 checks S_3, and so on. In other words, as we will discuss in the next section, the *bootstrapping* and *kernel* code contained in the system partitions already implements several mechanisms to maintain a *chain of trust*.

Based on this observation, BareDroid restores only S_1 (the *root* of the chain of trust), and the code contained in every partition of the $S_1 \rightarrow \ldots \rightarrow S_n$ chain will then verify the integrity of the other partitions.

Restoring User Partitions. User partitions are often modified by (both benign and malicious) Android apps. For this reason, the technique adopted for system partitions is inefficient, since the *integrity check* would fail often. A more efficient solution is to skip the integrity check and directly restore the user partitions before the analysis of each app.

Similarly to the previous case, restoring all user partitions is conceptually straightforward: one could just fully rewrite their content after the analysis of each application. However, this option is not efficient, especially because, depending on the device, the user partitions might be quite large (in some cases, up to few GBs).

To restore user partitions efficiently, we devised the following technique. For each user partition U_i, BareDroid maintains three different copies: the two working copies U_i', U_i'', and the clean copy $\hat{U_i}$. The two working copies are used as follows. While partition U_i' is used to analyze the given application, a background thread restores partition U_i'' to its clean state (starting from a clean copy, $\hat{U_i}$). Then, when the analysis of an application is over, the role of U_i' and U_i'' is swapped: the partition U_i'' is used to perform the analysis of the *next* application, while, in the meantime, the background thread restores U_i' to its clean state.

It is important to observe that, in principle, the integrity of the clean copy of each userdata partition, $\hat{U_i}$, needs to be verified before being used to restore U_i' or U_i''. In fact, even if there are no incentives for doing so, a malicious application that specifically targets our infrastructure could modify $\hat{U_i}$, and this could affect all subsequent applications' analysis. In this case, we ensure the integrity of this partition by relying on a kernel-level partition lockdown.

Kernel-level Partition Lockdown. We rely on a kernel-level mechanism (as discussed in the next section, in our current implementation we rely on SELinux) that ensures that the permissions associated to the $\hat{U_i}$ partition are set as *read-only*. This way, all write attempts to $\hat{U_i}$ are blocked. Clearly, for this mechanism to be effective, the assumption we made when discussing our threat model needs to hold. However, for the sake of completeness, in Section 6.2 we discuss several strategies for those situations in which one cannot assume to rely on such kernel-level mechanism. Finally, we note that, in principle, we could have applied the same protection mechanism for the system partitions as well. Nonetheless, we opted for not doing so: we believe that if a malicious application is able to successfully modify the `system` partition on a user's mobile device, it should have the same effect on BareDroid, so that the analysis result reflects the behavior of the malicious app accurately. Also, our system relies on a very efficient mechanism to perform the integrity check of the `system` partition, and thus the benefit of avoiding performing an integrity check is negligible.

4. IMPLEMENTATION

In this section we discuss many implementation details related to our system. First, we provide some technical details on how the boot process and the chain of trust are implemented in Android. Then, we will discuss the details related to how the `system` and `userdata` partitions are restored, how their integrity is verified, and how modifications are prevented. Finally, we discuss several aspects and technical details on how we used BareDroid to build an *in-house* cloud of Android devices, or a *phone cloud*.

4.1 Android Background

4.1.1 Android Partitions

Android uses several partitions to organize files and folders on a device. Each of these partitions plays a different role in the functionality of the device. The main partitions are the following (additional non-standard partitions can also be present depending on the phone's model):

- `aboot`: it contains the *bootloader*. The bootloader is the software component that is in charge of starting the boot process of the device. The bootloader is usually written by hardware vendors and typically starts the execution of the code in either the `boot` or `recovery` partitions (letting the user choose upon boot);
- `boot`: it enables the phone to boot. It includes the *kernel* and the *ramdisk*;
- `recovery`: it can be considered as an alternative `boot` partition that lets the user boot the device into a *recovery console* to perform advanced recovery and maintenance operations on it;
- `system`: it contains the entire operating system, including the kernel, the Android framework and user interface, as well as all the system applications that come pre-installed on the device;
- `userdata`: it contains all user's data, such as user contacts, messages, settings, and user-installed applications;
- `misc`: it contains miscellaneous system settings (e.g., the OEM lock/unlock switch). Conceptually, the content of this partition can be viewed as a low-level configuration file for all components involved in the booting process.

4.1.2 Android Boot Process and Chain of Trust

We now describe the high-level steps that constitute the boot process in Android. The first step of the Android boot process is the execution of the Boot ROM code, which then executes the *bootloader* in the `aboot` partition. The bootloader is a special program (separated from the Linux kernel) that is used to properly initialize the memory components and load the kernel to RAM. By default, the bootloader verifies that the contents of the `boot` and `recovery` partitions have been signed with one of the keys contained in the embedded keystore (e.g., OEM key). For verifying `boot` and `recovery` partitions, the bootloader attempts to verify the boot partition using the OEM key first and try other possible keys only if this verification fails.

Since BareDroid relies on a custom `boot` partition (to specify a proper SELinux policy) and on a custom `recovery` partition (that is in charge of *swapping* the role of the two copies of the userdata partitions) the bootloader tries the verification using the certificate embedded in the partition signature.

Typically, the execution moves from the *bootloader* to the `boot` partition. However, by pressing a specific combination of buttons during the boot (i.e., the power button together with the "Volume Up" button), a user can access to a boot menu that allows to start the device in "recovery mode", which executes the content of the *recovery* partition, and to modify the content of the different partitions using the USB connection. In addition, in most of the devices, by pressing a specific combination of buttons for a few seconds, a user can "hard-reboot" a device (i.e., force the hardware to reboot, independently from the status of the running software).

During the execution of `boot` partition's content, the kernel is responsible for setting up the verification of the `system` partition. Due to its large size, the `system` partition typically cannot be verified in the same way as the previous parts, but must instead be verified as it is "being accessed" through the *dm-verity* kernel mechanism[1]. *dm-verity* is a kernel driver that verifies each block read from the `system` partition against a hash tree created during the setup phase. The root hash is signed with a certificate stored in the boot image ramdisk. Note that this certificate can be trusted since it is verified by the *bootloader*, as described above.

When a *dm-verity* error is detected for a given block, an I/O error is raised and the block with the unexpected content is made inaccessible to user-space applications. Moreover, in this case, the device must be rebooted, and *dm-verity* must be started in logging mode during all subsequent restarts until any of the verified partitions is *re-flashed*. Through the adoption of *dm-verity*, one can be assured that the `system` partition can never be changed (or re-mounted in read/write mode), because these operations would change the superblock used to calculate the hash. For this reason, BareDroid uses the same mechanism to establish a chain of trust.

4.2 Restoring System Partitions

As we mentioned, we extensively use the *dm-verity* kernel-level mechanism to ensure that, at the beginning of the boot process, the `system` partition has not been tampered with during the analysis of a previous application. However, ac-

[1]https://source.android.com/devices/tech/security/verifiedboot/index.html

cording to our threat model an application could get root access and compromise the kernel. So, at least in principle, it might have a way to tamper with the *dm-verity* process itself. For this reason, just before the analysis of each application, BareDroid overwrites the content of the `boot` (that contains the implementation of the *dm-verity* mechanism) and the `recovery` partitions. This ensures that, at the beginning of each analysis, the system can rely on a non-compromised *dm-verity* mechanism (which we use as the root of our chain of trust), and it can then be used to determine whether the `system` partition has been tampered with (and thus should be *re-flashed*). Moreover, note that this is efficient because both these partitions are quite small.

4.3 Restoring User Partitions

As mentioned in the previous section, restoring the full `userdata` partition before each application analysis is not efficient enough, the reason being the large size of these partitions (on the 16GB model of the Nexus 5 device, the `userdata` partition size is about ∼13GB).

Instead, to speed up the restore process, BareDroid maintains three versions of the `userdata` partition (as explained in Section 3.3). One of them, *userdata_copy* acts as a *clean* snapshot of the partition's content. The other two, *userdata1* and *userdata2* play a different role for each analysis: when *userdata1* is used for the analysis of the current application, a background thread restores the *userdata2* partition by using the *userdata_copy* clean snapshot. For the analysis of the next application, the roles of the two partitions are swapped: *userdata2* is used for the analysis, while the background thread restores *userdata1*. As will be discussed in Section 5, this mechanism significantly improves the overall overhead of the restoring process.

4.4 Kernel-level Partition Lockdown

Although the mechanism described in the previous section significantly improves the performance of the restoring phase, a malicious application could tamper with the *userdata_copy* partition to indirectly affect the analysis of the subsequent applications. In fact, while this partition is initially mounted as read-only, a malicious application could simply re-mount it with read/write permission and modify it.

To mitigate the risk of the attacks described above, BareDroid leverages the use of SEAndroid [29]. SEAndroid is a project useful to implement a mandatory access control (MAC) model in Android, by using SELinux to enforce kernel-level MAC. SELinux policies are expressed at the level of security contexts. SELinux requires a security context to be associated with every process (or subject) and resource (or object), which is used to decide whether access should be allowed or not.

BareDroid uses a modified version of the AOSP SELinux policy to protect the integrity of files in the *userdata_copy* partition. Our policy is designed to specify which processes can mount and re-mount the partition. In our case, we specified BareDroid's update process as the only process that has *read* access and can re-mount the *userdata_copy* partition. Beyond that, writing to block devices, raw I/O, and mknod() are also locked down.

4.5 Phone Cloud

In the previous sections we have introduced all the technical details to enable the use of BareDroid on a device. However, the device represents just one of the required components to develop a bare-metal analysis infrastructure. For this work, we built an infrastructure (in fact, a cluster of phones) that is able to manage and coordinate in an automatic way all the steps described in Section 3.3. Our infrastructure currently comprises nine devices: eight Nexus 5 32GB with Android 5.1.0_r3, and one Asus Nexus 7 2012 (WiFi) 32GB using Android 5.1.0_r3.

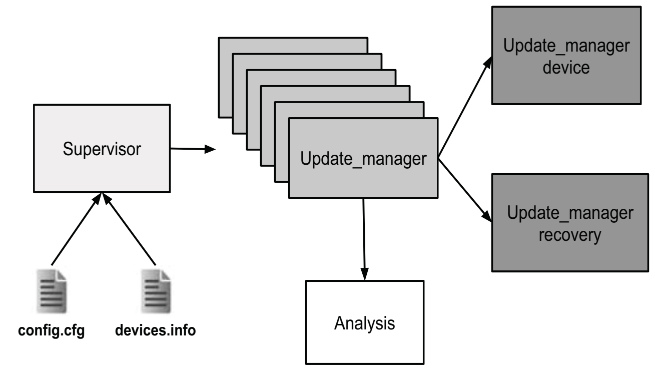

Figure 1: Architectural overview of the phone cloud infrastructure.

An orchestration software component, which we named *Supervisor*, manages and coordinates the analysis of multiple applications. For example, this component is in charge of rebooting and restoring a device after the analysis of each application. This component communicates with each phone through the Android Debug Bridge (ADB). ADB is a versatile command line tool that allows communicating with an Android device. Furthermore, the *Supervisor* is *physically* connected to the devices through a powered USB hub. In our experience, the USB hub needs to provide at least 500 mAh per port. During one of our stress-tests with a less powerful USB hub, the batteries of all phones got completely discharged.

Figure 1 shows an overview of the architecture of our phone cloud. The following are the components that constitute our system:

Supervisor: It constitutes the front-end of the infrastructure. It provides a command line interface (CLI) used to start/stop the analysis and monitor the status of the various devices;

config.cfg: It is a configuration file that contains information about the infrastructure (e.g., where to store logs);

device.info: It is a configuration file that contains information (e.g., DeviceId) about the devices used by the infrastructure;

Analysis: It provides a base class that can be used to implement specific malware analysis techniques.

Update_manager: Each device is managed by a dedicated *update_manager* process. It manages the workflow of the analysis and triggers the transitions between states. Depending on the state, the *update_manager* can start either one of the following two procedures:

Table 1: Time necessary by BareDroid to restore a device

Restoring step	Time (seconds)
restore the `recovery` partition using ADB	0.963
reboot into *recovery mode*	8.923
swap `userdata` partitions	1.976
boot the operating system	19.900
total	**31.762**
if dm-verity detects errors in the `system` partition:	
send `system` partition through ADB	27.927
rewrite `system` partition	35.233
total	**94.922**

1. *Update_manager_device*, which manages the update of the `userdata` partition;
2. *Update_manager_recovery*, which manages the swap of the `userdata` partitions when the device is in "recovery mode";

5. EVALUATION

In this section we will evaluate BareDroid under different aspects. First of all, we will measure the time our system needs to restore a device before starting a new analysis in the scenarios outlined in Section 3.2. Then, we will evaluate the cost-effectiveness of BareDroid when compared to both an emulator-based analysis system, and a system naively restoring a device at every reboot. In addition, we will demonstrate that BareDroid is resilient to current state-of-the-art approaches to detect emulators. Finally, we will show how existing real-world malware samples are unable to detect our system, allowing dynamic analyses on them to return more detailed and realistic results.

5.1 Performance Overhead

5.1.1 Device Restoring Time

For our experiments, we used an LG Nexus 5 device with Android 5.1.0_r3. All reported measures have been averaged on 5 runs of the experiment. Table 1 reports the time needed by all the steps performed by BareDroid to restore a device. In total, 31.762 seconds are necessary to restore a device.

As explained in Section 3.3, if one of the integrity verification steps fails, restoring the device obviously takes additional time. Specifically, if *dm-verity* detects corruption in the `system` partition, 63.797 additional seconds are required, bringing the total restoring time to 94.922 seconds.

We also tested the time needed to perform a full restore of a device, without using BareDroid. Recall that without using BareDroid the only feasible way to completely restore a device's state is to read using ADB and write on the device's flash memory a full *clean* copy of the `recovery`, `system`, and `userdata` partitions. This procedure requires 141.268 seconds. Thus, BareDroid's restore time is 4.44 times faster than the restore time of the vanilla approach.

Table 2: Values used in our cost analysis

	Cost per device/cpu	Restore time
BareDroid	$349	31.768s
full device restore system	$349	141.268s
emulator-based system	$300	1s

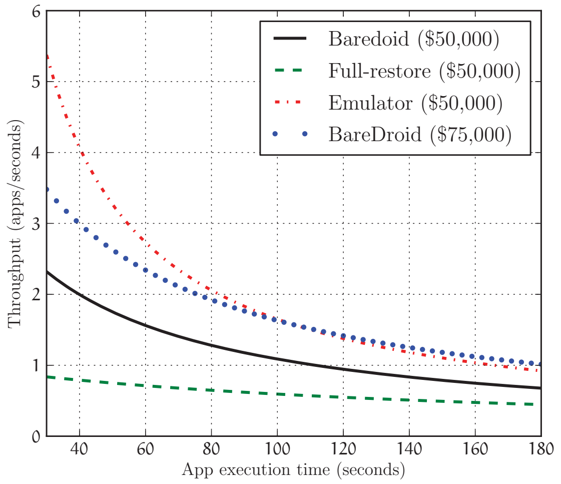

Figure 2: Cost analysis of using BareDroid and other approaches when performing large-scale malware analyses.

5.1.2 Analysis Slowdown

After the completion of the device's boot, BareDroid restores a copy of the `userdata` partition to speedup the next restore, as explained in Section 3.3. The time required by this step is 25.351 seconds. This part of the restoration happens in parallel with the sample analysis. Hence, this time overhead can be ignored if a sample-analysis run lasts longer than 25 seconds, which is usually the case in practice.

Moreover, we evaluated if the presence of an underlying process performing this step slows down an analysis running on the device. For this, we run the benchmarking app *AnTuTu* [2] while the restoring process was running, and while it was not. The benchmarking application did not report any significant difference in the two cases. The score reported by AnTuTu with and without the restore process is 28,353 and 28,355 respectively. For this reason, we believe that the performance impact caused by the restoring process is negligible.

5.2 Cost Analysis

Based on the speed and cost of one device, we evaluate and compare the throughput (in terms of apps per second) that can be achieved using BareDroid, an emulator-based system, and a system in which a full restore of a device is performed every time.

For this evaluation, we assumed that BareDroid is used with a Nexus 5 device (since this is the device we used to perform the experiments described in this section), with a cost per unit of $349 [5]. To evaluate the cost of an emulator-

[2]www.antutu.com

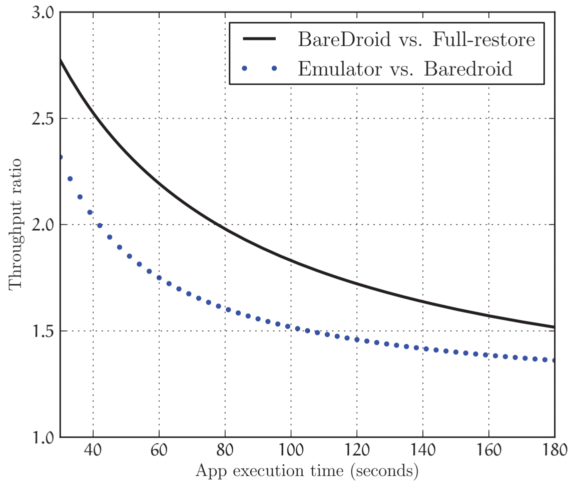

Figure 3: Throughput ratio between the different analyzed systems.

Table 3: Number of file operations generated by malicious samples in BareDroid and in an emulator-based system. The percentage between file operations detected by an emulator-based system and BareDroid is written in parenthesis. The * symbol indicates that the app crashed upon start, probably due to anti-emulator checks.

Sample	Emulator-based system	BareDroid
Android.HeHe.1	2 (11.76)%	17
Android.HeHe.2	9 (27.27)%	33
Android.HeHe.3	2 (11.76)%	17
Android.HeHe.4	0* (0.00)%	50
Android.HeHe.5	0* (0.00)%	50
Android.HeHe.6	9 (27.27)%	33
Android_Pincer.A	3 (8.82)%	34
OBAD.1	0* (0.00)%	32
OBAD.2	0* (0.00)%	32
total	**25 (8.39%)**	**298**

based system, we considered the price of a high-end server and we evaluated the cost per physical core. We then considered a scenario in which a physical core and 4GB of RAM are allocated for each emulator. This implies a cost of $300 [28] for each running emulator. Note that this is a conservative assumption, likely to overestimate the performance of an emulator-based system. In fact, apps run significantly slower in emulators with respect to when run on real devices and, for this reason, when using an emulator it is usually necessary to run a given app for a longer amount of time to achieve comparable results. Finally, we assume that in an emulator-based system, the restoring time is almost instant (1 second). Table 2 summarizes the parameters used in our cost analysis.

Figure 2 assumes a $50,000 equipment investment, and it shows the throughput in the three considered scenarios, given a desired analysis time. This evaluation shows how BareDroid achieves a performance that is in-between an emulator-based system and a system performing a full-device restore, while, at the same time, offers a realistic execution environment that cannot easily evaded by emulator-aware malware. Moreover, the cost overhead necessary to achieve the same throughput of an emulator-based system is only about 50% (as shown by the blue dotted line in Figure 2).

Assuming the same scenario, we also computed the ratio of the throughput obtained with BareDroid and a system performing a full restore at every reboot, and the ratio of the throughput obtained with an emulator-based system and BareDroid. Figure 3 reports the results. It is interesting to note how, for analysis times higher than two minutes, the ratio between the emulator and BareDroid is less than 1.5, whereas the ratio between BareDroid and a full-restore system is always higher than 1.5. We believe these results clearly show the practicality of using BareDroid.

That being said, it is important to note that our cost analysis has some limitations. For instance, it does not take into consideration the power consumption, the extra maintenance factors such as the human time spent to setup the BareDroid infrastructure (e.g., the setup of the *Supervisor* machine and the cabling), and the wear of the flash memory (the program-erase cycles of a NAND flash is typically about 100K). Another limitation of our cost analysis is that it does not take into account that malware might specifically target our infrastructure, for example by tampering with the *system* partition: in such case, BareDroid's restore time would be higher. Although we are currently not aware of any malicious sample with such capability, future malware samples could attempt to do so. However, while restoring the system partition takes more time, it is trivial to detect such tampering attempts, thus making this kind of advanced malware easily identifiable.

5.3 Security Evaluation

As discussed throughout the paper, one of the key limitations of modern analysis systems for Android apps is that they are based on emulators, and can therefore be easily detected and evaded. In this section, we will discuss the results of several experiments that highlight the resilience of BareDroid against these emulation-detection techniques.

We first tested the app developed as part of the paper describing Morpheus [16]. This app is designed to detect emulators via hundreds of different checks, and it represents the current state-of-the-art in emulation detection in Android. As expected, when run in BareDroid, this application did not detect it was being run within an analysis environment.

As a second experiment, we also tested the resilience of BareDroid against emulator-detection techniques used by real-world malicious samples: when these malware samples detect they are running within an emulator, they do not perform specific behaviors to avoid detection and hinder reverse engineering. To show that BareDroid improves the analysis of these samples, we designed a prototype analysis system, and we verified that this analysis elicits more behaviors when run in BareDroid compared to when it is run in an emulator-based system. It is important to note that while we believe many different malware analysis systems can be run on top of BareDroid, it would not be possible to run those analysis approaches that intrinsically rely on system emulation, such as taint analysis on native code.

Our analysis dynamically stimulates a sample (by using a manual approach) for 1 minute, and it monitors its behaviors. As described in [25, 31], one of the approaches to describe the behavior of Android malware is to log the system-calls invoked during the execution, and to then perform out-of-the-box behavioral analysis. To leverage this approach without introducing modifications to the kernel or adding specific components we take advantage of SELinux [29]: In practice, before the execution of each system call, the kernel queries SELinux to know whether a given process is authorized to perform the requested operation.

In particular, since, by default, SELinux records only the denied operations, we modified the SELinux policy to obtain information about each "granted" system call. The modification used to implement our analysis consists in the introduction of an *auditallow* rule per each *allow* rule in order to get a complete report of the operations performed by an app. From this report, we extracted information about file operations and we used it as an empirical measure of the amount of behaviors performed by the malicious application under analysis.

We run our experiment on several malicious samples that were known to be able to detect whether they are executed within an emulator. In particular, we considered the following applications:

- six variants of *Android.HeHe* [12]. This sample is able to steal text messages and intercept phone calls;
- one sample of *Android_Pincer.A* [10]. This sample is able to forward SMS messages and perform other actions based on commands it receives from a remote C&C server;
- two variants of *OBAD* [6]. This sample supports a wide variety of malicious features and communicates with a remote C&C server.

The results of our analysis are summarized in Table 3. Our results show that, globally, the emulator-based system detected only 8.39% of the file operations performed by the analyzed samples in BareDroid. These results show how emulator-based systems are ineffective in analyzing such samples. In addition, it is interesting to notice that 4 samples did not perform any file operation at all when executed in an emulator-based system.

6. DISCUSSION

6.1 Fingerprinting

As discussed in Section 2, an emulator can be detected using several techniques. Although it is possible to adopt a set of countermeasures for some of these techniques (e.g., checking for specific emulator's properties), it is extremely challenging, if not outright impossible, to correctly implement mitigations for some others (e.g., performance and timing issues). The key observation that makes this aspect problematic is the following: normal users do not use emulators, and hence the mere detection of the presence of an emulator is already enough for a malicious application to fingerprint and evade all existing analysis systems. We believe BareDroid significantly raises the bar when analyzing evasive malware applications. In fact, our infrastructure is constituted by the very same devices that a normal user would use on a daily basis.

That being said, we note that, even if BareDroid drastically reduces the fingerprinting surface by executing apps on a very common device class, ad-hoc fingerprinting techniques could still be used to detect the specific devices running BareDroid. For example, malicious applications could try to fingerprint a given analysis infrastructure by checking the device's MAC address or IMEI number, or by analyzing the partitions table. However, our proposed solution pushes malicious applications from attempting to fingerprint a device class (emulator vs. real device), to attempting to fingerprint a very specific device (e.g., one of the specific Nexus 5 phone we used). We believe this to be a much more challenging goal. In fact, mimicking a different instance of a specific device (another Nexus 5) is considerably easier than mimicking another device class (as emulators do). Moreover, fingerprinting the specific device can be in part mitigated by, for example, reusing some of the anti-evasion techniques employed by emulator-based dynamic analysis systems [22, 27, 30, 34], such as changing, for each run, the device's identifiers.

Nonetheless, we acknowledge that some advanced attacks (such as fingerprinting a specific device from its accelerometers' imperfections [8], USB charging state, the never-changing geographic position, the presence of extra partitions, or the process that restores the *userdata* partition) will be still effective against BareDroid, and that full undetectability is an open research problem.

6.2 Attacks against BareDroid

The implementation details described throughout this paper assume that kernel security functionalities (e.g., SELinux) are not compromised during an analysis. In this section, we elaborate on how an attacker getting around them could tamper with the BareDroid infrastructure, and we propose several countermeasures.

Most importantly, as long as malicious apps are not be able to tamper with the Boot ROM, the *bootloader*, the `recovery` partition code, and the *dm-verity* functionality that will gain control after reboot, we can exclude that a malicious app can persistently modify the content of most of the device's partitions. In other words, as long as the device reboots normally, the content of the following partitions is guaranteed to be untampered: `aboot` (checked by the Boot ROM), `boot` (restored at every reboot), `system` (checked by *dm-verity*), `recovery` (restored at every reboot), `userdata` (restored from an on-device copy at every reboot).

When considering a more aggressive threat model in which, for example, a malicious app could tamper with the SELinux module, other attacks are possible. For example, a malicious app could change the copy of the `userdata` partition that BareDroid uses during the device's restore procedure. To defend from this threat, one countermeasure would be to check the content of the copy of the `userdata` partition every a fixed amount of restores (as previously mentioned, it would be too time consuming to perform the check after each reboot), where the frequency of this check could be tuned depending on the security properties that a user of BareDroid desires. This check could be implemented in the `recovery` code by reading the content of this partition and verifying its hash. In case this verification process fails, a pristine copy of the `userdata` partition can then be copied again in the device (and the analysis's results performed in between the last two checks would need to be invalidated). In addition, an attacker could perform specific attacks against one of the devices used by BareDroid. For example, an attacker could tamper with the content of the `aboot` partition so that the verification step of this partition (performed by the Boot ROM) would fail: this, in turn, would make the entire device inoperable.

Even though these attacks can cause, in the worst case, a monetary loss, BareDroid would functionally act as a canary, signaling and detecting a very malicious app *early on*, and effectively preventing the end users' devices from getting damaged. In fact, BareDroid can easily detect these sophisticated attacks by noticing a failure in the restoring process of a device. Moreover, note that these attacks are possible only when an application can successfully exploit a vulnerability in the kernel: since these vulnerabilities are rare (and very valuable), we believe it is very unlikely that an attacker would *utilize* a Linux kernel zero-day, at the very high risk of being discovered, for the mere purpose of attacking a malware analysis infrastructure.

Finally, we consider the scenario in which we remove the assumption that the device is guaranteed to reboot normally: in other words, the attacker is able to "fake" the device reboot. In this scenario, all the considerations above are no longer valid. To deal with this specific attack, BareDroid would need to rely on a separate method (e.g., a machine that presses the appropriate button combination, as mentioned in Section 4.1.2), to hard-reboot the device: although it is not trivial to properly implement this mechanism, it would guarantee taking the device to a known state (i.e., the boot menu) and re-establish our chain of trust as previously exposed.

6.3 Alternative Implementations

Alternative implementations of BareDroid are possible. For instance, with larger modifications, it would be possible to boot a device using data and code stored in an external storage, such as a network shared drive, which can be restored easily and fast, similarly to how network boot and iSCSI technologies can be used in servers. Alternatively, specific hardware devices could be used to provide IPMI-like functionality, or rewrite the content of the flash memory without the need of using the USB connection or the code contained in the `aboot` partition.

If fully realized, these approaches would have the advantages of allowing a faster restore and avoiding entirely the need of on-device code to perform the device's restore (thus making completely impossible for an attacker to interfere with the restoring process). However, using hardware modifications, instead of commodity devices like those we used in BareDroid, has two important disadvantages: First, the cost of a large scale deployment of BareDroid would increase significantly; Second, using ad-hoc devices would increase the fingerprintability of BareDroid since they would likely introduce several discrepancies with respect to the commonly-used hardware. Nevertheless, we consider exploring the possibility of using different hardware devices as one very interesting direction for future work.

7. RELATED WORK

7.1 Sandbox Evasion

The problem of evasion in dynamic malware analysis is well-known. For desktop platforms, several techniques have been proposed to detect virtualized and emulated environments [3, 4, 11, 23, 26]. The main approach to detection is to find artifacts of the execution environment that are not present in a hardware-based environment. Initially, the detection techniques were focused on detecting the emulated or virtualized CPU [11, 23, 26]. Then, it has expanded to more generic approach of fingerprinting software, hardware, and external configurations of the analysis environment [4,17,35]. Even though these techniques are developed for desktop platforms, the core ideas are still related and applicable to Android analysis environment. Chen et al. first proposed a generic taxonomy of evasion techniques used by malware against dynamic analysis system [4]. The taxonomy proposes abstract groups based on where the artifacts originate from.

There are a few recent works on Android analysis environment detection [16, 24, 33]. Vidas et al. and Petsas et al. explored several analysis environment artifacts that are indicative of emulated environment [24, 33]. These environment artifacts include emulator-specific properties such as IMEI value, performance timing, properties of attached input devices, and other artifacts that are unique to the specific analysis environment. Jing et al. developed a system, called Morpheus, which is capable of automatically finding several thousands of such artifacts [16]. These artifacts can be used as heuristics to evade analysis environments.

7.2 Transparent Analysis

Sandbox evasion techniques have been frequently used by evasive malware in desktop platforms. Mobile malware is likely to follow a similar trend. For desktop platforms, many transparent malware analysis systems have been proposed to mitigate the problem of evasive malware [9, 14, 18, 32]. Cobra [32] proposed mitigation techniques for evasion of debugger-base analysis. Many analysis systems proposed out-of-the-box analysis approaches to improve transparency [1, 14, 27]. Ether [9] leveraged hardware-based virtualization technology to overcome emulator and software artifacts [9]. BareBox [18] proposed bare metal environment for transparent analysis and developed techniques to improve the scalability of the approach. Our approach, based on a bare-metal environment, is similar to BareBox. However, to the best of our knowledge, our work is the first to build an automated system of this kind for Android malware analysis. In fact, all current state-of-the-art Android analysis systems are based on emulators [2, 20, 25, 31, 34], which are known to be easily detectable.

8. CONCLUSIONS

Authors of malicious Android apps have started to fingerprint emulated analysis environments, as a mean to avoid detection from all popular Android malware detection engines. Despite this trend, apps are currently still being analyzed in emulated environment because executing them on bare-metal devices in scale has been unfeasible, as one can only analyze an order of magnitude less apps on bare-metal then on emulators at the same price-point for the hardware.

In this paper, we designed and implemented BareDroid, a system that makes the analysis of Android apps on bare-metal feasible at scale. We have shown that BareDroid is cost-effective (when compared to emulators) and, at the same time, cannot be evaded by emulator-aware Android malware. Moreover, we have shown that BareDroid is not being detected by the latest research on analysis-environment detection.

The goal of our work is to provide a platform on top of which existing and future analysis engines can perform malware detection without the risk of being evaded by the mere presence of an emulator-like environment. This is why, for the benefit of the security community, we release BareDroid as an open source project.

9. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers and our shepherd, William Enck, for their valuable feedback. This material is based on research sponsored by DARPA under agreements number FA8750-12-2-0101 and FA8750-15-2-0084. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The work was also supported by the Office of Naval Research (ONR) under grant N000140911042, the Army Research Office (ARO) under grant W911NF0910553, by the National Science Foundation under grant CNS-1408632, by Lastline, Inc., and Secure Business Austria. This work was also partially supported by a Google Research Award (winter 2014) and by the Italian Ministry of Research within the PRIN project "GenData 2020."

10. REFERENCES

[1] Anubis. `http://anubis.cs.ucsb.edu`.

[2] Sanddroid. `http://sanddroid.xjtu.edu.cn/`.

[3] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna. Efficient Detection of Split Personalities in Malware. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2010.

[4] 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.

[5] CNET. Google's $349 Nexus 5 hits today with LTE, KitKat. `http://www.cnet.com/news/googles-349-nexus-5-hits-today-with-lte-kitkat/`.

[6] Contagio mobile mini-dump. OBAD. `http://contagiominidump.blogspot.it/2013/06/backdoorandroidosobada.html` .

[7] DexLab. Detecting Android Sandboxes. `http://www.dexlabs.org/blog/btdetect`.

[8] S. Dey, N. Roy, W. Xu, R. R. Choudhury, and S. Nelakuditi. Accelprint: Imperfections of Accelerometers Make Smartphones Trackable. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2014.

[9] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2008.

[10] F-Secure. Android Pincer A. `https://www.f-secure.com/weblog/archives/00002538.html` .

[11] P. Ferrie. Attacks on Virtual Machine Emulators. Technical report, Symantec Corporation, 2007.

[12] FireEye. Android.HeHe. `https://www.fireeye.com/blog/threat-research/2014/01/android-hehe-malware-now-disconnects-phone-calls.html` .

[13] G. Ho, D. Boneh, L. Ballard, and N. Provos. Tick tock: building browser red pills from timing side channels. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, 2014.

[14] X. Jiang and X. Wang. "Out-of-the-Box" Monitoring of VM-Based High-Interaction Honeypots. *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, 2007.

[15] X. Jiang, X. Wang, and D. Xu. Stealthy Malware Detection through Vmm-based Out-of-The-Box Semantic View Reconstruction. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2007.

[16] Y. Jing, Z. Zhao, G.-J. Ahn, and H. Hu. Morpheus: Automatically Generating Heuristics to Detect Android Emulators. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2014.

[17] A. Kapravelos, M. Cova, C. Kruegel, and G. Vigna. Escape from Monkey Island: Evading high-interaction Honeyclients. In *Proceedings of the Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2011.

[18] D. Kirat, G. Vigna, and C. Kruegel. BareBox: Efficient Malware Analysis on Bare-Metal. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2011.

[19] D. Kirat, G. Vigna, and C. Kruegel. Barecloud: Bare-metal Analysis-based Evasive Malware Detection. In *Proceedings of the USENIX Security Symposium (USENIX)*, 2014.

[20] P. Lantz, A. Desnos, and K. Yang. DroidBox: Android Application Sandbox, 2012.

[21] S. Mutti, Y. Fratantonio, A. Bianchi, L. Invernizzi, J. Corbetta, D. Kirat, C. Kruegel, and G. Vigna. BareDroid Source Code. `https://github.com/ucsb-seclab/baredroid`.

[22] J. Oberheide and C. Miller. Dissecting the Android Bouncer. SummerCon, 2012.

[23] R. Paleari, L. Martignoni, G. Fresi Roglia, and D. Bruschi. A Fistful of Red-Pills: How to Automatically Generate Procedures to Detect CPU Emulators. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, 2009.

[24] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. Rage against the virtual machine: hindering dynamic analysis of Android malware. In *Proceedings of the ACM European Workshop on System Security (EUROSEC)*, 2014.

[25] A. Reina, A. Fattori, and L. Cavallaro. A system Call-Centric Analysis and Stimulation Technique to Automatically Reconstruct Android Malware Behaviors. In *Proceedings of the ACM European Workshop on System Security (EUROSEC)*, 2013.

[26] J. Rutkowska. Red Pill... or how to detect VMM using (almost) one CPU instruction. `http://invisiblethings.org/papers/redpill.html`, 2004.

[27] J. Security. JOE Sandbox Mobile. `http://www.joesecurity.org`.

[28] Server Direct. Server prices. `http://www.serversdirect.com`.

[29] S. Smalley and R. Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2013.

[30] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, and J. Hoffmann. Mobile-sandbox: Having a Deeper Look into Android Applications. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, 2013.

[31] K. Tim, S. Khan, A. Fattori, and L. Cavallaro. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2015.

[32] A. Vasudevan and R. Yerraballi. Cobra: Fine-grained Malware Analysis Using Stealth Localized-executions. *Proceedings of the IEEE Symposium on Security and Privacy*, 2006.

[33] T. Vidas and N. Christin. Evading Android runtime analysis via sandbox detection. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*, 2014.

[34] L. Weichselbaum, M. Neugschwandtner, M. Lindorfer, Y. Fratantonio, V. van der Veen, and C. Platzer. Andrubis: Android Malware Under The Magnifying Glass. Technical Report TR-ISECLAB-0414-001, iSecLab, May 2014.

[35] K. Yoshioka, Y. Hosobuchi, T. Orii, and T. Matsumoto. Your Sandbox is Blinded: Impact of Decoy Injection to Public Malware Analysis Systems. *Journal of Information Processing*, 2011.