Mirage: Toward a Stealthier and Modular Malware Analysis Sandbox for Android

Lorenzo Bordoni, Mauro Conti, and Riccardo Spolaor(✉)

University of Padua, Padua, Italy
lorenzo.bordoni@studenti.unipd.it,
{conti,riccardo.spolaor}@math.unipd.it

Abstract. Nowadays, malware is affecting not only PCs but also mobile devices, which became pervasive in everyday life. Mobile devices can access and store personal information (e.g., location, photos, and messages) and thus are appealing to malware authors. One of the most promising approach to analyze malware is by monitoring its execution in a sandbox (i.e., via dynamic analysis). In particular, most malware sandboxing solutions for Android rely on an emulator, rather than a real device. This motivates malware authors to include runtime checks in order to detect whether the malware is running in a virtualized environment. In that case, the malicious app does not trigger the malicious payload. The presence of differences between real devices and Android emulators started an arms race between security researchers and malware authors, where the former want to hide these differences and the latter try to seek them out.

In this paper we present Mirage, a malware sandbox architecture for Android focused on dynamic analysis evasion attacks. We designed the components of Mirage to be extensible via software modules, in order to build specific countermeasures against such attacks. To the best of our knowledge, Mirage is the first modular sandbox architecture that is robust against sandbox detection techniques. As a representative case study, we present a proof of concept implementation of Mirage with a module that tackles evasion attacks based on sensors API return values.

1 Introduction

In recent years, mobile devices like smartphones, tablets and smartwatches have spread rapidly, thanks to their portability and their affordable price. These devices became everyday multi-purpose tools and, consequently, a receptacle for personal information. Among mobile operating systems, Android is the leading platform, with a market share of 86% in 2016 [12], and it is growing as a new target for malware. The Android operating system uses a modified version of the Linux kernel, where each app runs individually in a secured environment, which isolates its data and code execution from other apps. The operating system mediates apps' access requests to sensitive user data and input devices (i.e., enforcing a Mandatory Access Control). Without any permission, an app can only access few system resources (e.g., sensors, device model and manufacturer) [3].

S.N. Foley et al. (Eds.): ESORICS 2017, Part I, LNCS 10492, pp. 278–296, 2017.
DOI: 10.1007/978-3-319-66402-6_17

Although malware could escalate privileges by exploiting vulnerabilities in the operating system, new threats arise also from apps that run unprivileged. Malware for Android often harms users by abusing the permissions granted to it. For example, malware can cause financial loss by leveraging features such as telephony, SMS and MMS, while with access to camera, microphone, and GPS it can turn a smartphone into an advanced covert listening device. Moreover, the leak of confidential data, such as photos, emails and contacts, threatens users privacy as never before [40]. Attackers usually spread malware infections by repackaging an app to contain malicious code, and by uploading it to Google Play (i.e., the official marketplace) or alternative marketplaces [42]. A possible approach to reveal malicious Android apps consists of analyzing them one at a time. However, this may be fighting a losing battle: Google Play counts more than 2.2 million apps today [32]. Thus, in recent years, researchers attention moved to the study of batch (i.e., non-interactive) analysis systems [17,34,35].

Malware analysts can examine suspicious apps through static analysis and dynamic analysis. On one hand, static analysis consists of inspecting the resources in the packaged app (e.g., manifest, bytecode) without executing it. Unfortunately, an adversary can hinder static analysis by using techniques such as obfuscation, encryption, and by updating code at runtime. On the other hand, dynamic analysis consists in monitoring the execution of an app in a test system. During such analysis, the sample (i.e., an app submitted for the analysis) runs in a sandbox. A sandbox is an isolated environment where malware analysts can execute and examine untrusted apps, without risking harm to the host system.

Academic and enterprise researchers independently developed many malware analysis systems for Android. For example, Google introduced Bouncer, a dynamic analysis system that automatically scans apps uploaded to Google Play [18]. Such analysis systems run long queues of batch analyses in parallel, and typically do not rely on real devices but on Android emulators. Unfortunately, emulators present some hardware and software differences (i.e., artifacts) with respect to real devices, which can also be recognized at runtime by apps: By detecting these artifacts, an app can easily recognize whether it is running or not on a real device. A malicious app can exploit emulator detection to evade dynamic analysis and show a benign behavior, instead of the malicious payload. Relying on such mechanism, malware authors might spread a new generation of malicious apps, which they would be hardly detectable with current dynamic analysis systems. While researchers keep improving dynamic analysis techniques, they are overlooking the accuracy of virtualization. In current malware analysis services for Android, the coarseness of the underlying emulator hinders researchers efforts.

Contribution. The contribution of this paper is a step towards the development of a stealthier malware analysis sandbox for Android, which reproduces as much as possible the characteristics of real devices. Our goal is to show malware the characteristics of an execution environment that appear to be real but are not actually there.[1] In this paper, we make the following contributions:

– We define six requirements to design a sandbox that can cope with current evasion attacks, and is easy to evolve in response to novel detection techniques.
– We propose Mirage, an architecture that fulfills all these requirements. Researchers can use Mirage to implement more effective malware analysis sandboxes for Android.
– We describe our proof of concept implementation of Mirage.
– We evaluate the effectiveness and the modularity of Mirage by tackling a specific and representative case: address sandbox detection techniques that exploit sensors capabilities and events.
– We show that Mirage, with our sensors module, can cope with most evasion attacks based on sensors that affect current dynamic analysis systems.

Organization. The rest of the paper is organized as follows. We start by presenting related work in Sect. 2. In Sect. 3, we define six requirements that we believe are essential to develop a malware analysis sandbox for Android. In Sect. 4, we present the components of Mirage. As a representative case study, in Sect. 5, we describe our proof of concept implementation of Mirage which addresses evasion attacks based on sensors. In Sect. 6, we compare our system with state of the art malware analysis services and we discuss its effectiveness in Sect. 7. Finally, Sect. 8 concludes the paper.

2 Related Work

Security researchers put a lot of effort in detecting PC virtualization [26,29]. However, in the era of cloud computing, a desktop or server operating system running inside a virtual machine is no longer a sign that dynamic analysis is taking place. Regarding mobile devices, nowadays malware analysts mainly rely on emulators, so malware can use emulator detection to evade dynamic analysis. Therefore, we strongly believe that evasion attacks on mobile emulators will be a hot topic for researchers in the years to come.

In what follows, we report the work related to the domain of sandbox detection. In [36], Vidas et al. described four classes of techniques to evade dynamic analysis systems for Android. The authors categorize such techniques with respect to differences in behavior (e.g., Android API artifacts, emulated networking), in CPU and graphical performances, in hardware and software components (e.g., CPU bugs), and in system design. Similarly, Petsas et al. in [28] presented evasion attacks against Android virtual devices. Jing et al. in [15] introduced Morpheus, a software that automatically extracts and rank heuristics to detect Android emulators. Morpheus retrieves artifacts from real and virtual devices,

[1] Like a mirage in a sand(box) desert, and this motivates the name of our proposed solution.

and it compares the retrieved artifacts to generate heuristics. Morpheus derived 10,632 heuristics from three out of thirty-three sources of artifacts. Maier et al. in [19] presented a tool for Android called Sand-Finger, which is able to collect information from sandboxes that malware can use to evade dynamic analysis.

Some industry presentations examined the sandbox detection problem as well. Strazzere, in [33], proposed detection techniques based on system properties, QEMU pipes and content in the device, which he embedded in an app for Android. Oberheide et al., in [24], and Percoco et al., in [27], showed that Google Bouncer is not resilient against evasion attacks, and an attacker can bypass it to distribute malware via Google Play marketplace. In addition to fingerprinting Bouncer, the former managed to launch a remote connect-back shell in its infrastructure.

Researchers proposed many dynamic malware analysis systems for Android that rely on an emulator. Few examples of such systems are CopperDroid [34], CuckooDroid [6], DroidBox [16] and DroidScope [41]. Other systems such as AASandbox [5], Andrubis [17], Mobile-Sandbox [31], SandDroid [30] and TraceDroid [35] perform dynamic analysis on an emulator as well, but they also use static analysis to improve their performances. In addition to performing both static and dynamic analysis, authors in [37] proposed to analyze samples using an emulator that they enhanced to tackle some evasion attacks. Although authors in [37] focus on how to perform malware analysis, it presents some interesting ideas against sandbox detection techniques. An interesting idea is to use a mixed infrastructure composed of real and virtual devices. Mutti et al. in [22] presented BareDroid, a malware analysis system based on real devices, instead of emulators, which consequently is more robust to evasion attacks. The authors estimated that a BareDroid infrastructure would cost almost two times the cost of a system based on emulators with the same capabilities. However, a virtual infrastructure is more elastic when compared to a cluster composed only of physical devices, which may suffer from under or over-provisioning.

To the best of our knowledge, the work by Gajrani et al. [10] is the most similar to our proposal. After giving a general overview on emulator detection methods, the authors present DroidAnalyst, a dynamic analysis system that is resilient against some of them. Their system can hinder evasion attacks based on device properties, network, sensors, files, API methods and software components. We share a common goal with the authors of [10]: the development of a malware sandbox for Android resilient against evasion attacks. However, we identified in [10] the following limitations (that we instead overcome with our proposal):

- Their analysis about artifacts in Android sensors API is not exhaustive. For example, they do not take some of our findings (see Sect. 5.2) into consideration.
- They propose a solution that consists of a set of patches to their analysis system based on QEMU. Therefore, it is not a general architecture like our proposal.
- DroidAnalyst uses an approach based on emulator binary and system image refinement, which does not allow the same emulator to impersonate two

different real devices, unless they are modified again and restarted. Conversely, the requirements of Mirage (presented in Sect. 3) discourage any modification to the emulator, since the sandbox would be less flexible and hard to maintain.

To evaluate the effectiveness of our sandbox detection heuristics based on sensors, we tried to submit to DroidAnalyst our sample, i.e., the *SandboxStorm app* (see Sect. 6). Unfortunately, the DroidAnalyst dynamic analysis subsystem was under maintenance, and is still not available at the time of writing.

3 Sandbox Requirements

After studying state of the art sandbox detection techniques [15,19,28,36], we define six key requirements that we believe are essential to develop a malware analysis sandbox for Android. Our goal is to derive the design of an architecture from the requirements, which can consist of one or more parts (i.e., components). We formulate the first three requirements on the basis of desired features to cope with the evasion attacks described in the aforementioned work (see Sect. 2). Moreover, we formulate three additional requirements taking into account that the sandbox should be flexible. The requirements are:

- **Stealthiness of sandbox components:** The components of the sandbox shall be unnoticeable by malware. Otherwise, an adversary could recognize a component of the sandbox, and evade dynamic analysis. This may seem a trivial requirement, but it serves as a cornerstone for our work. Nowadays, virtualized environments are not realistic and easily detectable [20,26,28,29]. Unfortunately, adding new countermeasures in such environments to achieve stealthiness produces new artifacts (e.g., processes and files). Such artifacts allow malware authors to fingerprint the whole system, causing the ineffectiveness of the countermeasures in place to make the virtualized environment stealthy. If the sandbox is not fully undetectable, it should be able to hide its imperfections, hiding them to the samples.
- **Consistency of bogus data:** The sandbox shall provide realistic and consistent information to the sample throughout the analysis. Otherwise, an adversary could detect the sandbox by exploiting the discrepancies in information that comes from different sources. To hide the artifacts in the emulator, the sandbox must produce a large amount of fake data. In this case, random generation is not an option, since it is prone to introduce discrepancies in data. For example, telephone numbers in contacts shall be composed of a country calling code plus a fixed number of digits [36]. A possible solution could be to use data collected from real mobile devices. In addition to that, the modules in the sandbox that inject such data must coordinate with each other to mimic a realistic environment.
- **Monitor known evasion attempts:** The sandbox should be able to notice whenever a sample is likely exploiting known detection techniques. Even if some artifacts are obvious but not fixable with nowadays technologies,

it is worth to log all the suspicious attempts and act in an alternative way. However, when an app looks for artifacts, it does not strictly means that that app is trying to evade the analysis.

- **Modularity of sandbox components:** The components of the sandbox shall be modular with respect to detection techniques that malware exploits. We believe this is a key requirement, since researchers keep reporting cutting-edge [15,19,28,36] evasion attacks every year. Researchers shall have the opportunity to develop, customize and publish new parts in a modular fashion, to keep up with the state of the art. A system designed to be open to new contributions makes it also improvable, in order to cope with emerging threats. Furthermore, since the Android operating system and its SDK change rapidly, sometimes new features break the compatibility with old ones that were available in previous versions. Hence, it is necessary to divide the components internally into modules. This allows to redesign and implement again just the modules that the changes affect.
- **No modifications to the Android source code:** The sandbox should not require any change of the Android source code. Although it would possible to alter APIs by modifying the operating system, compiling Android requires a significant amount of computational resources. In fact, a single build of an Android version newer than Froyo (2.2.x) requires more than two hours on a 64-bit consumer PC, plus at least 250 GB (including 100 GB for a checkout) of free disk space [1]. Even with the necessary resources and a semi-automated workflow, maintaining several versions simultaneously would be an overwhelming task.
- **No modifications to the Android emulator:** The sandbox should not require significant modifications to the emulator. Researchers are using different hypervisors and virtual machines for dynamic malware analysis, therefore we cannot focus on a specific technology. For example, systems like CopperDroid [34] use virtual machine introspection to reconstruct the behaviors of malware, hence such systems are potentially adaptable to any emulator. Forcing the scientific community to port the existing software to meet a modified emulator would likely lead to failure in the adoption.

4 Mirage: Our System Architecture

In this section we present Mirage, our architecture for a malware analysis sandbox robust against evasion attacks. One of the key feature of Mirage is that it is composed of processes that execute inside the operating system, and software that runs outside the emulator. This feature allows Mirage to be not tied to a specific analysis system.

In Fig. 1, we illustrate the four main components of Mirage which are the *Methods Hooking Layer* (Sect. 4.1), the *Events Player* (Sect. 4.2), the *Coordinator and Logger* (Sect. 4.3), and the *Data Collection App* (Sect. 4.4).

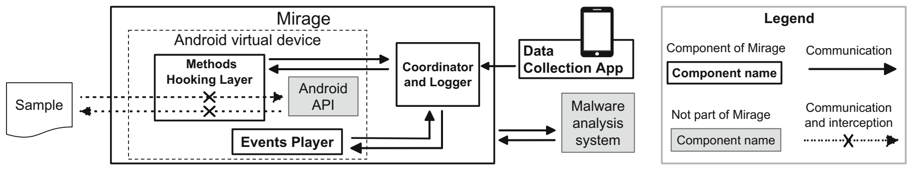

Fig. 1. Mirage architecture, highlighting its components and their interactions.

4.1 Methods Hooking Layer

The first component of Mirage architecture is the *Methods Hooking Layer*. This component executes as a process in the Android operating system. The main function of *Methods Hooking Layer* is to intercept calls to methods of Android API and manipulate their return value. Such manipulation occurs just whenever the original returned value may reveal the presence of the underlying emulator. Relying on this component, we can address the majority of behavioral differences. As an example, we can return a well-formed telephone number when a sample asks for `TelephonyManager.getLine1Number()`, instead of the default one (which in an emulator always begin with 155552155, followed by two random digits). Since it is possible to predict which artifacts the *Methods Hooking Layer* introduces, we can use such component to hide them as well. Moreover, hooked methods should perform minimal computation to reduce the risk of detection via computational timing attacks.

The code of the *Methods Hooking Layer* executes directly on a compiled operating system image. Hence, such code is debuggable without modifying and compiling every time the Android source code. In compliance with the modularity of sandbox components requirement (see Sect. 3), the modular sub-architecture of the *Methods Hooking Layer* makes it flexible with respect to changes. The *Methods Hooking Layer* divide hooks by target artifacts, thus they are editable without touching the other hooks. Moreover, such sub-architecture allows researchers to share their proof of concepts or mature modules in a common framework. However, system constants expose some artifacts as well (e.g., the ones contained in `android.os.Build`).

4.2 Events Player

Real mobile devices generate many events in response to external stimuli, hence hooking methods calls and manipulation their return value is not enough to simulate such asynchronous behavior. In order to make our runtime environment as realistic as possible, we need the *Events Player* replay recorded or generated streams of events in the emulator. Besides the touch screen, the main sources of events are sensors (e.g., accelerometer, thermometer) and multimedia interfaces (e.g., camera, microphone).

The *Events Player* replays tidily the streams of events, respecting their order. The accuracy of values domain is crucial to build a stealthy sandbox.

Indeed, the sandbox would be vulnerable to detection and fingerprinting, whether the injected events do not resemble the ones that come from a real sensor (e.g., they are out of range). Similarly to the *Methods Hooking Layer*, the *Events Player* uses only tools from Android, Android SDK and emulators, without requiring any modification.

4.3 Coordinator and Logger

The *Coordinator and Logger*, as it results clear from its name, has two roles: to coordinate and to log. Its first role as coordinator consists in ensuring consistency of bogus data, which the other components inject into the emulator. Whenever the *Methods Hooking Layer* loads a new module, or when the *Events Player* opens an events stream, we have to instruct the coordinator on how to manage such hooks or events stream in accordance with the other modules. A deep study of the interaction between Android features lead to a set of rules, which the coordinator feature is able to interpret. For example, data that sensors acquire is interdependent (e.g., accelerometer and GPS). Moreover, actuators on the device (i.e., the screen, the notification LED, the flash, speakers and the vibrator) can also influence data that sensors record (e.g., speakers may influence the microphone).

The second role of this component consists in logging what happens inside the sandbox. This logging feature of the *Coordinator and Logger* is useful to have an insight on which detection techniques the samples are probably exploiting. In addition to that, the logging feature is even more useful to signal whenever a sample attempts to use a known technique which the sandbox is not able to cope with yet. In this way, Mirage is able to monitor all possible evasion attempts. The *Methods Hooking Layer* reports to the *Coordinator and Logger* every suspect or evidence about the analyzed sample. The *Coordinator and Logger* could manage the analysis process entirely. As an example, this component could handle tasks such as sample submission or the presentation of results.

4.4 Data Collection App

The task of the *Data Collection App* is to collect information from real mobile devices. Then, the *Coordinator and Logger* will inject such information into the *Methods Hooking Layer* and into the *Events Player*. The goal of this process is to hide artifacts in the emulator. Indeed, acquiring data from different smartphones and tablets models allows to create emulator instances with different characteristics. At the same time, this approach also reduces the risk that malware authors detect a particular image. The app is also responsible of capturing events streams on the real device, and store them in a compact and easy to replay representation.

The *Data Collection App* can retrieve information from real mobile devices available in a laboratory, but a real advantage would be to collect data with crowd-sourcing. On one hand, in a laboratory scenario researchers could ask their colleagues or students to kindly give their help by installing the app and uploading data. Two examples of existing loggers for Android used for research purposes are DeviceAnalyzer [39] and DELTA [7]. On the other hand, in a crowd-sourcing scenario companies could include the *Data Collection App* in their mobile app. Adopting a freemium pricing strategy, companies can freely distribute their software for free in exchange for data collected from the device. With an app with a wide user base, it is also possible to acquire "disposable" data on demand. As an example, an antivirus app may offer to the user an extension of the license or a month of premium features, if she agrees to share with the company her sensors events for the next ten minutes. In both scenarios, we highlight that data collection must be respectful of the privacy of the participants, e.g., applying perturbation on collected data [11]. Such perturbation is meant to alter information in such a way that avoids to expose the contributing user's identity (e.g., biometrics, habits) and, at the same time, preserves the characteristics of the device.

5 A Representative Case Study: Tackling Evasion Attacks Based on Sensors with Mirage

In this section, we present the development process of a sensors module for Mirage, i.e., a collection of modules that emulates sensors in one or more Mirage components. Designing an effective countermeasure against evasion attacks requires a deep understanding of the problem. In this case study, we analyzed the differences in sensors characteristics between real devices and emulators. This case study has two purposes: (i) to briefly describe how we implemented Mirage, and (ii) to show that Mirage is effective against the proposed detection heuristics based on sensors. With a proof of concept implementation, we propose also an approach to carry out an investigation on evasion attacks. The final goal of such investigation is the development of a module for Mirage. In this way, researchers can extend Mirage to tackle novel evasion attacks, by following the workflow we present in this section.

In what follow, we discuss some choices about the components of our Mirage implementation. First, the *Methods Hooking Layer* rely on the Xposed framework as a methods hooking facility [38]. Xposed is an open source tool that allows to inject code before and after a method call. It is worthy of note that other hooking tools, such as Cydia Substrate [8], adbi [21] serve the same purpose. In particular, we preferred Xposed because Cydia Substrate is not open source and adbi supports only the instruction sets of ARM processors. Xposed by its nature is detectable, since it introduces some artifacts. However, subverting methods hooking detection techniques is not difficult, as pointed out in [4]. Secondly, the *Events Player* relies on a Telnet console in QEMU, which allows to remotely inject sensors events into the emulator. During our preliminary studies, we considered multiple alternative approaches. Unfortunately, most of the alternative approaches we investigated are not viable due to our requirements in Sect. 3 (e.g., modifications to the emulator) or because they are not compatible with recent Android versions (e.g., RERAN [14]). Although this is a QEMU-specific feature, other emulators (e.g., Genymotion, Andy) offer a similar events injection mechanism. Finally, we develop a custom *Data Collection App* and we implement the remaining components as a set of scripts.

The case study we report in this paper is focused on sensors artifacts. We chose detection techniques based on sensors for three reasons:

1. Researchers pointed the feasibility of such detection techniques [28,36] without providing any effective countermeasure.
2. A possible countermeasure against such detection techniques involves multiple components in our system (i.e., the *Methods Hooking Layer*, the *Events Player*, the *Coordinator and Logger*, the *Data Collection App*).
3. Accessing motion, position and environmental sensors do not require any permission. This means that the sensors-based detection techniques are stealthier than the ones that do not rely on sensors. In fact, a popular app can be repackaged to include a sensors-based detection technique, without altering the original permission list in its manifest.

Our workflow starts with threat modeling (described in Sect. 5.1), continues with artifacts discovery and analysis (Sect. 5.2), and ends with the implementation of the module (Sect. 5.3). By following the above steps, researchers can progressively improve Mirage, toward an ideally undetectable sandbox.

5.1 Threat Model

In our threat model, we assume an attacker that is running a malicious app on a mobile device, with full access to the Android sensors API. The sensors API is composed of `SensorManager`, `Sensor`, and `SensorEvent` classes, plus the `SensorEventListener` interface. An instance of `SensorManager` corresponds to the sensor service, which allows to access to the set of sensors available on the device. An instance of `Sensor` is related to a specific sensor, which can be hardware or software-based. The methods of the `Sensor` object permit to identify sensor capabilities. The `SensorEvent` class represents a single sensor event, that contains: the sensor type, the sensor state (i.e., value and accuracy), and the event timestamp. The `SensorEventListener` is a Java interface to implement in order to receive notifications whenever a sensor state changes. In our threat model, we also assume that the malicious app has a limited timespan before deciding whether to execute the payload or to remain dormant. In that time interval, the malicious app can monitor some sensors events.

5.2 Artifacts Analysis

Artifacts are imperfections that make a sandbox distinguishable from a real device. To put ourselves in attacker's shoes, we studied the Android sensors API in order to find out which sensors artifacts malware could leverage to evade dynamic analysis. First, we analyzed real smartphones such as LG/Google Nexus 5 and 5X, Samsung Galaxy S5 and S6, Galaxy Ace Plus, and Asus ZenFone 2. These real devices were running different operating system versions, ranging from Android 2.3 (API level 9) to Android 7 (API level 24), which is the most recent release at the time of writing. Then, we analyzed how emulators supports sensors. In this analysis, we considered Android SDK's emulator and Genymotion (free plan), given their popularity among developers. On one hand, the Android SDK provides a mobile device emulator based on QEMU (QEMU from now on). Such emulator uses Android Virtual Device (AVD) configurations to customize the emulated hardware platform. On the other hand, Genymotion is a third party emulator, but it is compatible with Android SDK tools. Genymotion allows developers to control features like the camera, the GPS and battery charge levels. Most of the features of Genymotion are also manageable through a Java API [13].

The first discrepancy we noticed is that both emulators support a limited set of sensors. The developers of Android defined some types of sensors (i.e., the ones whose names begin with `android.sensor.*`). For such sensors, the `getType` method returns an integer number less than or equal to 100. Moreover, vendors can introduce custom sensors, i.e., the sensors for which `getStringType` returns a string that begins with `com.google.sensor.*` in the Nexus 5X. Given this fact, we can argue that a malware author who wants to target as much users as possible will not rely on device-specific sensors. In addition to that, malware authors have to focus on sensors available in API level 9 in order to target most of the devices (approximately 99.9% of the active devices according to Google Play [2]).

In our analysis, we considered the sensors embedded in real devices and the ones simulated by virtual devices. For each sensor, we called all methods available in the `Sensor` class. As an example, in Table 1 we show the discrepancies in terms of return values for accelerometer methods on real and emulated Nexus 5X. In Table 1, we also include the return values for our proposal, which we discuss in details in Sect. 6. Malware authors can rely on those discrepancies to develop simple detection techniques (a single conditional statement is enough). We refer to these techniques as *static heuristics*, since they exploit an artifact due to the Android API, which is not related to events streams. The accelerometer, thanks to its wide availability, is particularly well suited for broad-spectrum heuristics.

Table 1. Example of return values for Nexus 5X accelerometer in real devices, vanilla emulators (i.e., QEMU and Genymotion) and QEMU enriched with Mirage.

Device	getName	getVendor	getFifoMax-EventCount
Real	`BMI160 accelerometer`	`Bosch`	`5736`
QEMU	`Goldfish 3-axis Accelerometer`	`The Android Open Source Project`	`0`
Genymotion	`Genymotion Accelerometer`	`Genymobile`	`0`
QEMU + Mirage	`BMI160 accelerometer`	`Bosch`	`5736`

In the literature, researchers already pointed out the feasibility of *dynamic heuristics*, in which they exploited sensors events that emulators generate [28,36]. We investigated further: for each real mobile device at our disposal, we registered callback methods to receive changes in sensors state. By applying the option `SENSOR_DELAY_FASTEST`, we got those states as fast as possible. In our experiments, we observed that collecting an incoming stream of events for ten seconds is enough for our purpose. We collected sensors data from real mobile devices in three different scenarios: lying on a table, while typing and leaving them in a pocket while walking. Then, we repeated the data collection task on QEMU and Genymotion emulators. Such emulators allow only two modes of screen rotation: portrait and landscape.

During our experiments, we were able to observe some differences between real and emulated motion sensors. In real mobile devices, we noticed that motion sensors (e.g., the accelerometer) quickly oscillate among a small range of values, even when the device is lying on a flat surface. In emulators, we noticed that it is possible to stimulate the accelerometer by changing from landscape to portrait mode. In contrast, without rotating the screen, each motion sensor in emulators produce the same value. Table 2 records the constant values that each sensor in QEMU produces. It is worthy of note that some sensors in QEMU produce values only along one axis, so in Table 2 we mark the cells related to the other two axes as n/a.

Table 2. Constant values produced by sensors in QEMU, grouped by screen orientation.

getStringType		Portrait			Landscape		
		values[0]	values[1]	values[2]	values[0]	values[1]	values[2]
`android.sensor.`	`accelerometer`	`0`	`9.77622`	`0.813417`	`9.77622`	`0`	`0.813417`
	`magnetic_field`	`0`	`0`	`0`	`0`	`0`	`0`
	`light`	`0`	n/a	n/a	`0`	n/a	n/a
	`pressure`	`0`	n/a	n/a	`0`	n/a	n/a
	`proximity`	`1`	n/a	n/a	`1`	n/a	n/a
	`relative_humidity`	`0`	n/a	n/a	`0`	n/a	n/a

To show the detectability of the analyzed emulators, we implemented a fast dynamic heuristic that observes the variance of accelerometer values. Since by default such emulators are able to produce at most two different accelerometer values along one axis, if the accelerometer produces at least three different values it is likely to be on a real device. In general, static heuristics are faster than dynamic ones, because static heuristics do not require looping or waiting. Hence, the execution time of our dynamic heuristics depends on how fast sensors generate events, since it needs to retrieve at least three values in order to decide. Unfortunately, dynamic heuristics that rely on sensors are harder to tackle than static ones. Indeed, an ideal countermeasure against such dynamic heuristics consist in simulating or replaying events.

5.3 Module Implementation

In order to tackle the evasion attacks in Sect. 5.2 with Mirage:

- We included in our *Data Collection App* the code we used for artifacts analysis.
- We patched the discrepancies in return values using information we obtained from the *Data Collection App*.

To address static heuristics, we added to the *Methods Hooking Layer* our knowledge about the characteristics of real sensors. In fact, the *Methods Hooking Layer* can intercept methods calls directed to the `Sensor` class, returning values that we collected from sensors of a real device. Xposed executes a method before (pre-method) and after (post-method) each method hooked [38]. The pre-method can evaluate and alter the arguments, or it can return a custom result. In our implementation, we used only post-methods. In fact, first we allow the original methods to execute, then we inspect the sensor type, and finally we alter its return value accordingly. After defining an hook for each method of `Sensor` class, for every available sensor type, Mirage is able to mimic a real device.

In our proof of concept implementation, we leveraged QEMU to develop the replay mechanism of *Events Player*. This is because QEMU exposes a console via Telnet and it supports more sensors than Genymotion. Such console allows to control the virtualized environment, including sensors. The syntax of a Telnet command is `telnet <host> <console-port>`, where the default port is 5554. Once connected, we can set the values for a given sensor using the command `set <sensorname> <value-a>[:<value-b>[:<value-c>]]`. We implemented a prototype that reads a stream of values from a file and injects such stream (i.e., replay) into a running emulator. Under these settings, the *Coordinator and Logger* ensures that the *Events Player* replays for each sensor a sequence that is part of the same stream. This solution is adaptable to all Android emulators that expose a similar injection mechanism (including the premium releases of Genymotion), and it does not require any modification to the emulator.

6 Evaluation

For the evaluation of our proposal, we developed the *SandboxStorm app*. Such app includes the static and dynamic heuristics in Sect. 5.2, thus it easily detected both QEMU and Genymotion emulators. To show that similar artifacts are also present in state of the art systems, we submitted our *SandboxStorm app* both to offline and online malware analysis services. We picked CuckooDroid and DroidBox as offline dynamic analysis software, mainly because they are open source. CuckooDroid adds to the Cuckoo Sandbox a QEMU-based virtual machine to execute and analyze Android apps [6]. DroidBox relies on QEMU and it tries to understand the sample's behavior by repackaging the app with monitoring code [16]. Then, we picked some state of the art online malware analysis services from [23]. Among them SandDroid [30] and TraceDroid [35] were in working order. Moreover, we had the opportunity of testing the *SandboxStorm app* also on Andrubis [17] before its shutdown. Unfortunately, CopperDroid [34] was stuck on a long queue of unaccomplished analysis at the time of our evaluation.

In Table 3, we summarize the results we obtained by running our *SandboxStorm app* in the aforementioned malware analysis systems. The results show that both static and dynamic heuristics of our *SandboxStorm app* successfully detected the presence of an underlying emulator. In the worst case, our dynamic heuristic took about 74 seconds to detect that the app is running on a virtual device. However, we believe that such amount of time is still negligible in this scenario, since a malicious app can delay the start of its malicious behavior by 74 seconds. It is worthy of note that Andrubis and TraceDroid did not made available any sensor. However, the absence of sensors is a clear evidence that the sample is not running on a real device.

In order to evaluate our sensors module, we executed the *SandboxStorm app* in QEMU enhanced with our proof of concept implementation of Mirage. Once distributed, our *Data Collection App* can retrieve more information from real smartphones and tablets, in order to build several profiles. The static heuristics in *SandboxStorm app* failed to detect Mirage, because its *Methods Hooking Layer* manipulates the return values of methods on the fly. In Table 1, we compare some artifacts in vanilla emulators (see Sect. 5.2) to the corresponding values we collected during the execution inside Mirage. These values are identical to the ones provided by a real Nexus 5X. Moreover, the methods in `Sensor` class return realistic values for each sensor in the emulator. Even our dynamic heuristic in *SandboxStorm app* failed to detect the sandbox. In fact, the *Events Player* injects previously recorded sensors values, that are naturally different from the ones that we described in Table 2. Overall, *SandboxStorm app* is not able to notice that it is running in Mirage.

We also tested the stealthiness of our Mirage implementation against DroidBench (version 3.0-develop) [9], an open source benchmark suite, which includes several test cases (i.e., apps) for emulator detection. In particular, we tested our sensors module against "Sensors1" test case. The "Sensors1" test successfully recognized our real devices and both vanilla QEMU and Genymotion, but it missclassified our QEMU enhanced with Mirage as a real device.

Table 3. Detection time by analysis system and heuristic type.

Analysis system	Static heuristic	Dynamic heuristic
CuckooDroid	16 ms	70361 ms
DroidBox	18 ms	69581 ms
Andrubis	16 ms	n/a
SandDroid	15 ms	73964 ms
Tracedroid	17 ms	n/a

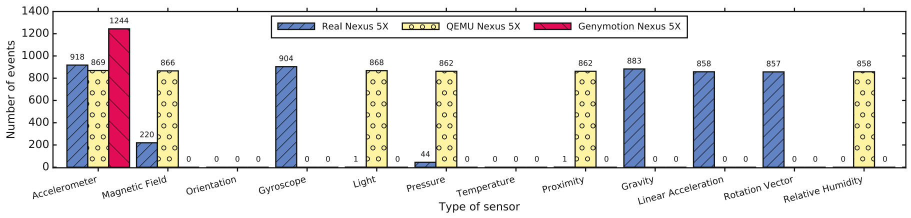

Fig. 2. Number of events retrieved in ten seconds for several sensors and devices.

7 Discussion

The core contribution of our work is Mirage, a framework that allows researchers to build malware analysis systems for Android that are robust against evasion attacks. Mirage is potentially adaptable to work with any emulator or hypervisor. Our implementation of Mirage, is able to tackle complex heuristics based on sensors of our *SandboxStorm app* (see Sect. 6). In the rest of this section, we first discuss the effectiveness of our sensors module and possible limitations of Mirage. Then, we present some alternative implementations of the *Events Player*. Finally, we remark the importance of modularity in the design of Mirage.

Effectiveness of the Sensors Module. We verified that the sensors module is effective against our static and dynamic heuristics based on sensors (see Sect. 6). We further investigated the reasons underneath the success of our implementation of Mirage against the “Sensor1” heuristic of DroidBench. We noticed that such heuristic checks the presence of 13 distinct types of sensors. If DroidBench detects a number of sensors that is less than or equal to a given threshold (the authors set this threshold to 7), it reports the presence of an emulator. Hence, this test case consists in a static heuristic. In Mirage, the *Methods Hooking Layer* is responsible to cope with detection techniques that are purely static. Since our sensors module for the *Methods Hooking Layer* can fake the presence of sensors that are not actually there, “Sensor1” test case fails to count the number of real sensors.

Overall, the development process of the sensors module for Mirage helped us to show that our proposal can be a useful tool to tackle evasive malware on Android. Unfortunately, such module has some shortcomings. Given a specific real device simulated by Mirage, the *Methods Hooking Layer* is able to mimic static characteristics of sensors available in such device, even if these sensors are not present in the underlying emulator. Similarly, Mirage can also hide the sensors that are available in the emulator whenever they are not present in the real device. Nevertheless, for sandbox detection techniques that monitor the events stream (like the dynamic heuristic in *SandboxStorm app*), our *Events Player* implementation is limited to the set of sensors supported by the underlying emulator (i.e., QEMU, in our current implementation).

Pre-filter NDK-based Applications. The Native Development Kit (NDK) allows embedding native code into Android apps. NDK can be useful for developers that need reduced latency to run computationally intensive apps (e.g., games) or to reuse code libraries written in C and C++. Unfortunately, allowing developers to code using NDK enables mobile malware authors to develop kernel-level exploits and sophisticated detection techniques [28,36]. Malware that is able to measure performances at low level (e.g., that measure the duration of time-consuming computation) can evade analysis systems based on virtualization by performing computational timing attacks. This is because such systems insert additional layers between the Android operating system and the CPU, with respect to real devices. Even though all these kind of artifacts are hard to patch, we can easily detect the usage of native code. Since Mirage cannot handle NDK-based malware properly, it could forward these samples to a real device or to a small bare metal infrastructure for the analysis. We assume that most of the requests are addressed in our sandbox, and we consider the forwarding of the samples to a real device as a last chance.

Alternative Implementations of the Events Player. Before deciding to rely on the Telnet console in QEMU in order to implement the sensors module in the *Events Player*, we considered different approaches. In order to simulate sensor events in real time, researchers in [28] suggested to use external software simulators, like OpenIntents Sensor Simulator (OISS) [25], or to adopt or a record-and-replay approach, like RERAN [14]. On one hand, OISS is an app that transmits simulated or recorded sensors streams to an emulator. Unfortunately, to receive the generated sensors events, OISS forces apps developers to use its own API instead of Android sensors API. This constraint is unsuitable for malware analysis, because the source code of the sample usually is not available. On the other hand, RERAN is a tool that first captures an events stream from a real device and then injects the stream in another device. Input events are recorded from `/dev/input/event*` in the source device and stored in a trace using *getevent* tool of Android SDK. A custom replay agent reads the trace and writes events to `/dev/input/event*` in the destination device. Unfortunately, in recent smartphones (e.g., Nexus 5X, Galaxy S5) *getevent* tool is able to get the touchscreen and buttons events, but not sensors ones.

Modularity of Mirage Components. One of the most important lesson we learned during our experiments is that the Android platform is rapidly and unpredictably changing. To give a significant example, while we were testing our heuristics, the developers of Android released an improved version of QEMU (along with Android Studio 2.0 release). This new version handles many more simulated events than the previous ones, actually resembling a real device. To show that, in Fig. 2 we compare the number of events retrieved in ten seconds from real and virtual Nexus 5X. In this experiment, we used the last releases of QEMU and Genymotion. Each sensor that the two emulators support is able to generate a number of events approximately equal or greater than the sensors on the real Nexus 5X. Unfortunately, the developers of Android arbitrarily decided to remove the opportunity to set sensors values via Telnet in the improved version of QEMU, which we exploited in our implementation of the *Events Player*.

Although we still do not know if the developers will reintroduce such feature in the future, this change highlights that the modularity in Mirage components is fundamental. Now QEMU is able to produce a significant number of sensors events on its own. Hence, it is possible to hook also methods of `SensorEventListener` class and manipulate the returned sensors values directly (without injecting sensors events from the *Events Player*). The isolation between the modules of the *Events Player* and the *Methods Hooking Layer* allows to relocate the simulation of sensors events from the former to the latter, without modifying the other modules. Nonetheless, to give a more comprehensive proof of concept of Mirage, we preferred to use the previous release of QEMU (prior to Android Studio 2.0), keeping the simulation of sensors events in the *Events Player*.

8 Conclusion

In this paper, we take a step towards the stealthiness of malware analysis sandboxes for Android. After carefully reviewing the state of the art, we enlisted six essential requirements that an analysis system have to fulfill to tackle evasion attacks. Hence, we proposed Mirage, a framework that fulfills all these requirements. In this paper, we also presented a representative case study, which shows how Mirage can cope with sandbox detection techniques that exploit artifacts in emulators due to sensors API. To evaluate our proposal, we developed a proof of concept implementation of Mirage, enabled with our sensors module. To compare our sandbox to state of the art dynamic analysis services for Android, we also developed the *SandboxStorm app*. This app contains some static and dynamic heuristics to detect emulators, based on our findings about sensors API artifacts. Our thorough evaluation shows that all dynamic analysis systems that we tested are detectable by our *SandboxStorm app*. Conversely, Mirage resembled a real device and, consequently, sensors-based heuristics in *SandboxStorm app* and in DroidBench were not able to detect Mirage as a sandbox.

Acknowledgments. Mauro Conti is supported by a Marie Curie Fellowship funded by the European Commission (agreement PCIG11-GA-2012-321980). This work is also partially supported by the EU TagItSmart! Project (agreement H2020-ICT30-2015-688061), the EU-India REACH Project (agreement ICI+/2014/342-896), and by the projects “Physical-Layer Security for Wireless Communication”, and “Content Centric Networking: Security and Privacy Issues” funded by the University of Padua. This work is partially supported by the grant n. 2017-166478 (3696) from Cisco University Research Program Fund and Silicon Valley Community Foundation. This work is also partially funded by the project CNR-MOST/Taiwan 2016-17 “Verifiable Data Structure Streaming”.

References

1. Android. Building requirements. goo.gl/7rLNfX (2016)
2. Android. Dashboards. goo.gl/7ygJx (2016)
3. Android. Developer’s guide. goo.gl/lvtCmr (2016)

4. Bergman, N.: Android anti-hooking techniques in Java. goo.gl/vN1iDU (2015)
5. Bläsing, T., Batyuk, L., Schmidt, A.-D., Camtepe, S.A., Albayrak, S.: An Android application sandbox system for suspicious software detection. In: IEEE MALWARE (2010)
6. Check Point Software Technologies LTD. Automated Android malware analysis with Cuckoo Sandbox. goo.gl/pDokqw (2016)
7. Conti, M., Santo, E.D., Spolaor, R.: DELTA: data extraction and logging tool for Android (2016). arXiv preprint: arXiv:1609.02769
8. Freeman, J.: Instrument Java methods using native code. goo.gl/1yqeFj (2016)
9. Fritz, C., Arzt, S., Rasthofer, S.: DroidBench. goo.gl/MEPCsD (2016)
10. Gajrani, J., Sarswat, J., Tripathi, M., Laxmi, V., Gaur, M., Conti, M.: A robust dynamic analysis system preventing sandbox detection by Android malware. In: ACM SIN (2015)
11. Ganti, R.K., Ye, F., Lei, H.: Mobile crowdsensing: current state and future challenges. IEEE Commun. Mag. **49**, 32–39 (2011)
12. Gartner. Gartner says five of top 10 worldwide mobile phone vendors increased sales in second quarter of 2016. goo.gl/X0ArDi (2016)
13. Genymotion. Using Genymotion Java API. goo.gl/zCTuDl (2016)
14. Gomez, L., Neamtiu, I., Azim, T., Millstein, T.: Reran: timing-and touch-sensitive record and replay for android. In: IEEE ICSE (2013)
15. Jing, Y., Zhao, Z., Ahn, G.-J., Hu, H.: Morpheus: automatically generating heuristics to detect Android emulators. In: ACM ACSAC (2014)
16. Lantz, P.: Dynamic analysis of Android apps. goo.gl/bFvjWS (2015)
17. Lindorfer, M., Neugschwandtner, M., Weichselbaum, L., Fratantonio, Y., Van Der Veen, V., Platzer, C.: Andrubis-1,000,000 apps later: a view on current Android malware behaviors. In: IEEE BADGERS (2014)
18. Lockheimer, H.: Android and security. goo.gl/fFFQcC (2012)
19. Maier, D., Protsenko, M., Müller, T.: A game of droid and mouse: the threat of split-personality malware on Android. Comput. Secur. **54**, 2–15 (2015)
20. Matenaar, F., Schulz, P.: Detecting Android sandboxes. goo.gl/0fp4bB (2012)
21. Mulliner, C.: The Android dynamic binary instrumentation toolkit. goo.gl/bzvBzm (2016)
22. Mutti, S., Fratantonio, Y., Bianchi, A., Invernizzi, L., Corbetta, J., Kirat, D., Kruegel, C., Vigna, G.: BareDroid: large-scale analysis of android apps on real devices. In: ACM ACSAC (2015)
23. Neuner, S., Van der Veen, V., Lindorfer, M., Huber, M., Merzdovnik, G., Mulazzani, M., Weippl, E.: Enter sandbox: Android sandbox comparison (2014). arXiv preprint: arXiv:1410.7749
24. Oberheide, J., Miller, C.: Dissecting the Android Bouncer. SummerCon (2012)
25. OpenIntents. Sensor Simulator. goo.gl/n1a9XD (2014)
26. Paleari, R., Martignoni, L., Roglia, G.F., Bruschi, D.: A fistful of red-pills: how to automatically generate procedures to detect CPU emulators. In: USENIX WOOT (2009)
27. Percoco, N.J., Schulte, S.: Adventures in BouncerLand. Black Hat USA (2012)
28. Petsas, T., Voyatzis, G., Athanasopoulos, E., Polychronakis, M., Ioannidis, S.: Rage against the virtual machine: hindering dynamic analysis of Android malware. In: ACM EUROSEC (2014)
29. Raffetseder, T., Kruegel, C., Kirda, E.: Detecting system emulators. In: Garay, J.A., Lenstra, A.K., Mambo, M., Peralta, R. (eds.) ISC 2007. LNCS, vol. 4779, pp. 1–18. Springer, Heidelberg (2007). doi:10.1007/978-3-540-75496-1_1

30. SandDroid. An automatic Android application analysis system (2014). http://sanddroid.xjtu.edu.cn/
31. Spreitzenbarth, M., Freiling, F., Echtler, F., Schreck, T., Hoffmann, J.: Mobile-sandbox: having a deeper look into Android applications. In: ACM SAC (2013)
32. Statista. Number of apps available in leading app stores as of June 2016. goo.gl/tCnPXW(2016)
33. Strazzere, T.: Dex education 201 - anti-emulation. goo.gl/jrqaaJ (2013)
34. Tam, K., Khan, S.J., Fattori, A., Cavallaro, L.: CopperDroid: automatic reconstruction of Android malware behaviors. In: NDSS (2015)
35. Van Der Veen, V., Bos, H., Rossow, C.: Dynamic analysis of Android malware. Internet & Web Technology Master thesis, VU University Amsterdam (2013)
36. Vidas, T., Christin, N.: Evading Android runtime analysis via sandbox detection. In: ACM ASIACCS (2014)
37. Vidas, T., Tan, J., Nahata, J., Tan, C.L., Christin, N., Tague, P.: A5: automated analysis of adversarial Android applications. In: ACM SPSM (2014)
38. Vollmer, R.: XposedBridge development tutorial. goo.gl/P0piK (2016)
39. Wagner, D.T., Rice, A., Beresford, A.R.: Device analyzer. In: Proceedings of ACM HOTMOBILE (2011)
40. Wheatstone, R.: Pippa Middleton's iCloud hacked. goo.gl/xnNQ5u (2016)
41. Yan, L.K., Yin, H.: DroidScope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In: USENIX Security (2012)
42. Zhou, Y., Jiang, X.: Dissecting Android malware: characterization and evolution. In: IEEE SP (2012)