STDNeut: Neutralizing Sensor, Telephony System and Device State Information on Emulated Android Environments

Saurabh Kumar*, Debadatta Mishra, Biswabandan Panda, and Sandeep K. Shukla

Indian Institute of Technology, Kanpur, India
{skmtr,deba,biswap,sandeeps}@cse.iitk.ac.in

Abstract. Sophisticated malware employs various emulation-detection techniques to bypass the dynamic analysis systems that are running on top of virtualized environments. Hence, a defense mechanism needs to be incorporated in emulation based analysis platforms to mitigate the emulation-detection strategies opted by malware. In this paper, first we design an emulation-detection library that has configurable capabilities ranging from basic to advanced detection techniques like distributed detection and GPS information. We use this library to arm several existing malware with different levels of emulation-detection capabilities and study the efficacy of anti-emulation-detection measures of well known emulator driven dynamic analysis frameworks. Furthermore, we propose STDNeut (**S**ensor, **T**elephony system, and **D**evice state information **Neut**ralizer) – a configurable anti-emulation-detection mechanism that defends against the basic as well as advanced emulation-detection techniques regardless of which layer of Android OS the attack is performed on. Finally, we perform various experiments to show the effectiveness of STDNeut. Experimental results show that STDNeut can effectively execute a malware without being detected as an emulated platform.

Keywords: Android · Malware · Security · Sandbox · Emulation-Detection

1 Introduction

Mobile platforms like Android are common in modern-day devices because of its open-source availability and robust support for mobile application (App) development. According to a recent report published by International Data Corporation (IDC), the global market share of Android OS was 85.1% [16] in the year 2018. As a consequence of such a large scale adoption of Android and ever-increasing contributions in the Android App space, the security of these devices has become a non-trivial challenge recently. A study related to malware activities in the Android platform published by G DATA shows that in the first half of 2018, more than 2 million new Android malware were recorded. In other words, an Android malware is born in every seventh second [3].

* Corresponding author.

Existing approaches that address the security issues arising due to the rapid growth of Android malware can be broadly classified into two categories: static analysis based techniques and dynamic analysis/detection based techniques [30]. Techniques that are based only on static analysis [11,20,38] are insufficient to address the security issues presented by the malware especially designed to bypass the static analysis based defenses. For example, advanced malware employ techniques such as dynamic code loading, native code exploitation, Java-reflection mechanisms, and code encryption to bypass static analysis based detection [33]. In order to address the limitations of static analysis techniques, dynamic analysis techniques are preferred. While dynamic analysis is widely used, the existing frameworks fall short in tackling platform sensing malware which is a reality today as emulator-based analysis platforms are used as opposed to real devices for cost-effectiveness.

The problem: Many malware [36] employ techniques to detect the underlying emulation platform before showing their true behavior. To the best of our knowledge, none of the existing emulator driven dynamic analysis frameworks make claims regarding their effectiveness towards nullifying possible emulation-detection adopted by malware.

One of the root causes of the problems related to emulation-detection is heavy usage of emulated platforms by dynamic analysis solutions. Many dynamic analysis systems (Droidbox [21], MobSF [24], CuckooDroid [34], DroidScope [40], CopperDroid [32], and Bouncer [22]) are based on virtualized environments to perform malware analysis by executing the Apps in a controlled environment and collect various event logs for further analysis. As a negative consequence, malware developers utilize various emulation-detection techniques to detect the underlying execution environment and adapt their behavior accordingly.

Identifying the underlying execution environment by a malware is shown to be possible by many previous studies [18,25,27,36]. Recently, it has been shown that the dynamic analysis performed for identifying malicious Apps by the Google Bouncer (a dynamic analysis system deployed on Play Store) [22] can be bypassed by detecting the underlying execution environment [25,27]. Vidas et al. [36] present generic emulation-detection approaches that can be used to evade dynamic analysis, whereas Morpheus et al. [18] show more than 10000 heuristics to detect underlying emulated platforms. In contrast, DroidBench-3.0 [8], an open test suite for evaluating the effectiveness of an analysis system includes a subset of small Apps (based on [36]), which can help in analyzing the effectiveness of dynamic analysis frameworks. However, the emulation-detection mechanisms used by DroidBench-3.0 Apps are very basic and many dynamic analysis frameworks (e.g., CuckooDroid [34], Droidbox [21]) have already incorporated anti-emulation-detection measures in their design. Even though the dynamic analysis frameworks are capable of providing defense mechanisms against rudimentary emulation-detection, *malware developers find new ways to detect the underlying execution environment* at runtime [35]. For example, Google Play Protect which is used to certify Android Apps, fails to detect malware that spread across 85 different Apps affecting nine million Android devices [26].

Our goal: We believe, a dynamic analysis system should provide a configurable anti-emulation-detection mechanism, so that a smart malware developer would find it difficult to evade the dynamic analysis by studying the analysis framework. Moreover, we would like to emphasize the need for a validation mechanism to understand the effectiveness of the same.
Our approach: As a validation mechanism, we design a pluggable emulation-detection library with configurable levels of emulation-detection capabilities, which can be incorporated by any malware. We use this library to arm several existing malware with different levels of emulation-detection capabilities and study the efficacy of anti-emulation-detection measures of well known dynamic analysis frameworks. Further, using the findings of our analysis, we develop STDNeut (**S**ensor, **T**elephony system, and **D**evice state information **Neut**ralizer), a detailed anti-emulation-detection system fully designed using Qemu [9] based Android emulator [5].

A robust and extensible validation framework can provide the basis for understanding the effectiveness of existing dynamic analysis systems w.r.t. their anti-emulation-detection measures. Moreover, the framework should provide guidance principles for designing new dynamic analysis systems with detailed anti-emulation-detection measures. Towards these objectives, our contributions are as follows:
(i) We design an emulation-detection library encompassing several advanced detection techniques like distributed detection and GPS information (Section 3.1). We use this library to perform an empirical evaluation of existing dynamic analysis frameworks against the basic and extended emulation-detection techniques (Section 3.2). The library can be configured with varying levels of emulation-detection methods and can be embedded into different malware in a seamless manner.
(ii) We propose STDNeut by using the insights of the empirical validation of existing frameworks (Section 4) that remain undetected even if the emulation-detection is performed at any layer of the Android OS w.r.t. sensors, telephony system and device state. Further, we show the effectiveness of STDNeut in neutralizing different emulation-detection techniques (Section 5). Note that, detection of an analysis framework by observing a rooted device or the existence of an instrumentation framework like Xposed is beyond the scope of this paper.

2 Background and Related Work

In this section, we discuss the Base Transceiver Station (BTS) as a smartphone frequently communicates with it. After that, we provide a brief overview of the emulation-detection followed by the related work.

2.1 Base Transceiver Station

BTS [23] is a piece of wireless communication equipment that establishes communication between a mobile device and a network. The BTS is associated with

a base station ID that uniquely identifies a BTS worldwide. Base station ID comprises of four components: *(i)* mobile country code (MCC), *(ii)* mobile network code (MNC), *(iii)* location area code (LAC), and *(iv)* a cell ID (CID). A combination of these gives a unique identity to a BTS. Several commercial and public services are available which provide the geo-location of a cell by submitting its station's unique ID.

2.2 Emulation-detection

The primary issue with an emulated system is its inability to replicate a complete system that matches the exact configuration and characteristics of a physical device. The core idea of emulation-detection is to observe the differences between virtual and physical machines using a program to identify the underlying infrastructure. Vidas et al. [36] and Morpheus [18] have shown that such differences can be used to detect underlying emulated platforms through a stand-alone App. Vidas et al. [36] propose a few generic detection methods based on the device characteristics, e.g., differences in hardware components (like sensors and CPU information) and software components (like Google's Apps are not present). Morpheus [18] presents more than 10000 heuristics to detect the underlying emulated platform which has broadly classified it into three categories viz. i) Files, ii) APIs, and iii) System Properties related detection which are similar to the techniques proposed in [36]. The emulation-detection methods shown in [18,36] fall in the category of basic emulation-detection, and most of the dynamic analysis systems are capable of bypassing them.

2.3 Related Work

Static analysis techniques fail to capture the precise characteristics of an App because of the advanced App development techniques like dynamic code loading and reflection [33]. This has led to the introduction of a dynamic analysis of Android Apps. Dynamic analysis techniques execute an App in a controlled environment called "sandbox" which can be a real device or an emulated platform to observe its behavior. Dynamic analysis on a real device is costly and incurs significant overhead [31]; hence, an emulator driven sandbox gets the attention of security researchers.

Emulation driven analysis tools must provide the ability to hide the emulated environment from the target App along with the profiling features. In the absence of such defense mechanisms, an App can evade the dynamic analysis by detecting the emulated platform [18,25,27,36].

The techniques in [25,27] use API based detection and IP address based detection to evade dynamic analysis on Google Bouncer, which essentially works by determining whether the IP address belongs to Google or not.

DroidBench [8], a recent work, provides a set of Apps to detect the underlying virtual environment based on the methods proposed in [18,36]. Further, DroidBench also introduces some new methods that utilize the call history and number of contacts in emulation-detection. Similarly, Caleb Fento [13] and

Gingo [14] have developed stand-alone Apps to detect Android virtual devices. Caleb Fento [13] uses the information shown by Vidas et al. [36] as a detection method to detect Google's Android emulator. On the other hand, Gingo [14] extends the same to detect custom virtual devices (like Genemotion, Nox player) along with Google's Android emulator.

Other than the stand-alone Apps discussed so far, some android libraries [7,17,19] have also been developed to detect emulated Android devices which can be integrated with any App. Libraries [7,17] use similar information as presented in [14], whereas the library [19] uses the accelerometer data in the detection mechanism.

Additionally, Diao et al. [12] proposed an approach to evade runtime analysis by differentiating a user from a bot by analyzing the interaction pattern. This detection technique is inclined to differentiate user from a bot to bypass runtime analysis and does not focus on the emulation-detection.

Costamagna et al. [10] have shown the evasion of Android sandbox through the fingerprinting of usage-profile. This technique works by observing the device usage information like SMS, call history and others which remain the same when multiple samples of a malware family execute inside a sandbox. However, the information received at the server from numerous Apps (malware sample of the same family) during different executions remains identical. The same information is fed to the next subsequent malware sample to evade the dynamic analysis.

Existing sandboxes [21,24,34,40] provide some anti-emulation-detection measures to mitigate the emulation-detection attack. For example, DroidBox [21] modifies the Android Open Source Project (AOSP) to bypass the emulation-detection, while some others [24,34] utilize the hooking framework (like Xposed [39]) and provide static but realistic information. Though, they can defend against the basic emulation-detection in DroidBench but they do not work in the context of extended emulation-detection.

Some other anti-emulation-detection works have also been proposed that modify the targeted App before submitting it for analysis [29,37]. Siegfried et al. [29] use the backward slicing method and remove the emulation-detection related checks from an App. On the other hand Droid-AntiRM [37] performs bytecode instrumentation to defeat the emulation-detection. In both approaches, an App needs modification before submission for dynamic analysis. Thus, the integrity of an App is lost through such changes.

3 Motivation

To study the effectiveness of the existing dynamic analysis frameworks, we require a tool with varying levels of the emulation-detection method. In this section, first, we give an overview of the flexible emulation-detection library that we have designed with a collection of emulation-detection methods beyond the basic detection techniques (see Section 2.2). We use this library to evaluate the existing frameworks about their anti-emulation-detection measures empirically. At last, we present the insights learned from this evaluation in designing STDNeut.

Table 1. Classification of emulation-detection techniques.

Detection Categories	Description
Unique device information (basic)	Detection by observing unrealistic device information values (e.g., IMEI value is 00000)
Unique device information (smart)	Detection based on fixed reading of unique device information (e.g., IMEI value is constant)
Sensors reading	Absence of sensor or observing static values from fluctuating sensors (e.g., fixed reading of `Light` sensor)
Device State information	No change to the device state w.r.t. telephony signal, battery power.
GPS information	No change on GPS location data or fake location change
Distributed detection	Observing identical unique information for multiple devices in a network.

3.1 Overview of Emulation-detection Library (EmuDetLib)

As a validation mechanism, we have developed a flexible **emu**lation-**det**ection **lib**rary (EmuDetLib). The detection techniques in EmuDetLib can be broadly classified into five categories (refer Table 1): *(i)* Unique device information (UDI), *(ii)* Sensors reading, *(iii)* Device state information, *(iv)* GPS information, and *(v)* Distributed detection.

Unique device information: This method uses information like IMEI (international mobile equipment identity) and IMSI (international mobile subscriber identity), that is unique to a device and employs basic and smart methods to detect an emulated environment. In basic detection, EmuDetLib observes any unrealistic data (not in the prescribed format) obtained from the device, whereas in smart detection, the library also checks whether the information is static or not by comparing it against known static values of different frameworks.

Sensors reading: Nowadays smartphones have various sensors for different purposes that can be broadly classified into two categories—motion sensors and environmental sensors. As the data observed on these sensors fluctuate continuously, this insight can be used to detect the underlying emulated environment. A recent example of sensor-based detection is the observation of TrendMicro, where malware (in Play Store) make use of the motion-detection feature to evade the dynamic analysis [35]. This method detects an emulated environment by utilizing sensors count and/or by observing fixed sensor values from fluctuating sensors.

Device state information: In reality, a device state gets changed due to some internal/external event such as change in telephony signal strength, battery power and incoming SMS/Calls. However, such state changing behavior is missing in an emulated environment. Our library observes these information to detect the underlying emulated environment.

Using the GPS location information: GPS is also a sensor and malware can use similar methods (as explained above) that are used for other sensors to detect emulated platforms. However, the emulation-detection based on the GPS is somewhat different from other sensors, as explained below.

Android provides rich APIs to perform various tasks. One such API gives the power to generate a mock location that can be used by an App to introduce a fake location when queried. An Android App requires `ACCESS_MOCK_LOCATION` permission to use the mock location API. The other source for geo-location is BTS ID. Android provides API to query BTS ID, and we can get its geo-location

by using publicly/commercially available services (`https://opencellid.org`). Hence, the geo-location-based emulation-detection technique only works when one of the following conditions is satisfied: *(i)* there is no change in the geo-location of the device, *(ii)* the mock location API is used to set the geo-location of the device, or *(iii)* BTS geo-location is not collaborating with the GPS location.
Distributed emulation-detection: Nowadays, most Apps require communication with a centralized server to share their status or get new information. To identify a device uniquely at the server, an App typically generates a unique ID called an AppID. A smartphone also contains device-related unique IDs namely IMEI, IMSI, SIM Serial number, and others. These information can also help in identifying a device uniquely at the server as explained below.

It is trivial to see that a slightly different malware in terms of its signature can be generated easily by changing its package name, altering the function name and variable naming convention, or by introducing dummy code while retaining the overall functionality and the server address. Such malware can communicate the unique device information to a remote server to identify the emulated environment remotely. In this situation, the emulation-detection can happen at the server by querying the device information from the connected devices. If a server detects that multiple devices have identical information (expected to be unique), it can flag those devices as emulated environment. As this emulation-detection is carried out in the context of multiple connected devices, we classify this detection technique as a distributed emulation-detection.

The emulation-detection methods in EmuDetLib discussed above are configurable and any App can change its detection mechanism by creating a suitable configuration file. For more details on EmuDetLib and ethical concern, refer weblink: `https://skmtr1.github.io/EmuDetLib.html`.

3.2 Evaluation of Existing Frameworks

To perform empirical evaluation of the existing dynamic analysis frameworks, we have integrated EmuDetLib into the DroidBench-3.0 [8] benchmark Apps (referred to as EmuDetLib-Bench). Apart from the EmuDetLib-Bench, we have collected 1000 malware where the dex date is of the year 2019 from AndroZoo [4] along with the motion sensor's malware disclosed by the Trend Micro [35] (referred to as RealMal) to evaluate the existing frameworks. We have considered CuckooDroid [34], Droidbox [21], and MobSF [24] along with the vanilla Android emulator (referred to as emulator) [5] as the candidate analysis systems for the empirical study, as they are readily available. We exclude the online analysis systems and other sandboxes in this study. The main reason is that an online analysis system has a long waiting queue and takes a longer time to schedule a sample for the evaluation. Hence, these frameworks are not preferred for this evaluation.

Further, to evaluate GPS information based detection and distributed detection, we need a different environment. For GPS, we require a fake GPS location generation app inside an emulated device. For distributed detection, we

Table 2. Evaluation of existing framework against detection library EmuDetLib.

Detection Type	Sub-type	Emulator	Droidbox	CuckooDroid	MobSF
Unique Device Information	Basic	✓	×	×	×
	Smart	✓	✓	✓	✓
Sensors	Count	×	×	×	✓
	Reading	✓	✓	✓	✓
Device State	–	✓	✓	✓	✓
GPS	Cond *(i)* (Normal)	✓	✓	✓	✓
	Cond *(i)* (Fake)	×	×	×	×
	Cond *(i)* & *(ii)*	✓	✓	✓	✓
	Cond *(i)* & *(iii)*	✓	✓	✓	✓
Distributed (Server config)	No Emulation	×	×	×	×
	W/- Emulation	✓	✓	✓	✓

Note: ✓ represents successful detection of underlying emulation environment, whereas × represents failure in detecting emulation environment. We use this notation in the rest of the tables. In GPS based detection, "Fake" represents a sandbox executing fake GPS location generating App/service. Normal represents without fake location App/service, and rest of the condition is evaluated with both the setting i.e. fake and without fake app. In distributed emulation, no Emulation represent the server without emulation-detection algorithm whereas W/- Emulation represent server deployed with emulation-detection algorithm.

need a server where the emulation-detection method is deployed and requires multiple instances of the same sandbox running at the same time. We utilize the command and control server of the real malware Dendroid [28] by employing the emulation-detection algorithm (see Algorithm 2 at weblink: `https://skmtr1.github.io/EmuDetLib.html#al2`).

Table 2 shows the evaluation result of the emulation-detection of candidate sandbox against all the detection methods shown in Table 1. In Table 2, the sub-type represents the subcategory/configuration of the evaluation. As shown in Table 2, in distributed detection, when the server is configured with the emulation-detection method, none of the frameworks can hide their emulated environment. Similarly, in GPS-based detection, only with a fake app installed emulated-platform can bypass the detection mechanism in condition 1 (refer to GPS information based detection). In other cases of GPS, the sandbox is flagged as an emulated platform by the detection library. There is one other case in the sensors category with count where MobSF is the only sandbox that cannot bypass the detection mechanism. The reason being, it is designed on top of VirtualBox and does not support sensors. In contrast, all other sandboxes use an Android emulator, which comprises of 7-8 sensors inbuilt and bypasses the emulation-detection based on sensor count.

Similarly, on executing samples of RealMal (see Table 2 at weblink: `https://skmtr1.github.io/EmuDetLib.html#tl2` for classification and evaluation), Android SDK emulator cannot hide its emulated environment against malware samples with emulation-detection capability. Simultaneously, other sandboxes get detected by the malware samples under the category of device state and sensors. To reason about such behavior, we have investigated BatterySaverMobi malware from RealMal samples (see Listing 1 at weblink: `https://skmtr1.github.io/EmuDetLib.html#cs1` for code snippet), which uses accelerometer (line 5) reading to observe motion on a device. If any motion takes place, then it executes the malicious code (line 15). Hence, Such malware can bypass the dynamic analysis job performed on existing sandboxes.

3.3 Summary of Emulation-detection

Some key observations regarding the effectiveness of anti-emulation-detection measures of the existing analysis platforms against EmuDetLib are shown below.
i) Existing analysis frameworks are able to bypass the basic emulation-detection techniques based on unique device information. However, they fail to defend when the emulation detection attacks are performed by analyzing the underlying defense mechanism. The main reason being either the data is unrealistic (basic detection) or the data is realistic but static (smart attack).
ii) Each framework fails to defend against the emulation-detection attacks based on fluctuating sensors and GPS data since the data does not represent the realistic behavior of a device.
iii) Similar to the detection methods based on UDI, existing frameworks are also not able to defend against distributed emulation-detection. The observation of similar data for unique device-related information across multiple devices helps in raising the red-flag regarding the underlying emulated environment.
iv) Detection methods based on the device state (e.g. Telephony, Battery Power) also successfully detect the underlying emulated environment due to the absence of defense mechanisms in the analysis frameworks.

In short, the extended emulation-detection techniques show that the existing publicly accessible dynamic analysis frameworks do not provide foolproof anti-emulation-detection measures. Therefore, there is a need for a robust anti-emulation-detection approach that can hide the underlying platform from smart emulation-detection measures. Note that the emulation-detection techniques can also utilize the timing channel to detect the emulated platform (like timing measures against the graphics subsystem). Such heuristics require a sufficient number of events to understand the underlying execution environment, which tends to increase their code footprints and flag such an App as abnormal. Due to this limitation, we do not discuss any timing channel based emulation-detection methods. In the next section, we discuss the design and implementation of STDNeut which incorporates a robust anti-emulation-detection system.

4 STDNeut: Design & Implementation

In this section, first we discuss the process of generating realistic sensor's data and the challenges associated with it. After that we provide an overview of STDNeut, a detailed anti-emulation-detection system and elaborate on the design of its various components. STDNeut aims to neutralize emulation-detection using different sensors, telephony system, and device state data.

4.1 Realistic Sensor Data Generation

A smartphone contains multiple sensors (e.g., accelerometer, GPS, and others) or interacts with an external entity like BTS. A malware can use these sensors to detect an emulated environment. To nullify the effect of sensors based emulation-detection, we have identified three main challenges as follows:

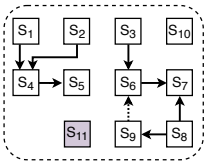


Fig. 1. An example of sensor's dependency graph. Sensor S_{11} in shaded box represents a new sensor introduced in the system.

(i) Existing sensors value should fluctuate with respect to time.

(ii) Detection of emulation environment through sensor correlation.

(iii) Model should be flexible to incorporate new sensors and sensor relations.

To better understand these challenges, let us take a directed graph shown in Figure 1 that represents eleven sensors (S_1 to S_{11}), and influence of one sensor on others in terms of driving the sensor's values. An arrow from sensor S_i to sensor S_j denotes that the value of sensor S_j depends on the value of sensor S_i. If we see an update in the value of sensor S_i, then sensor S_j's value should also be seeking an update according to S_i's value. As shown in Figure 1, some sensors do not depend on other sensors (sensor with zero in degree); we name them as independent sensors, whereas sensors with in degree ≥ 1 are called dependent sensors because the value of these sensors depends on the value of others.

Challenge *(i)* is easy to understand, which states that the value of the sensor should fluctuate concerning time. For example, let us consider sensor S_{10} (assuming as a light sensor) in Figure 1, the value of this sensor should be updated according to the operating environment lighting condition. Similarly, other sensor's value should also be updated w.r.t. time or working environment condition.

To understand challenge *(ii)*, consider two sensors S_4 (assuming as GPS) and S_5 (assuming as BTS). As shown in Figure 1, sensor S_5's value depends on the value of sensor S_4. This dependency is based on the distance between the values of S_4 and S_5, which cannot be more than x meters. This x may vary depending on the area density (population and obstacles) of the BTS. Further, to be more clear about challenge *(ii)*, let us include two more sensors S_1 (as time) and S_2 (as an accelerometer). The value of sensor S_4 depends on both the sensors, i.e., S_1 and S_2. If we consider time and GPS, then there is a correlation between the current GPS location and the previous location w.r.t. time elapsed. For example, if the current GPS location is Washington DC, a person cannot reach New York in five minutes. Similarly, when considering accelerometer and GPS, then the measurement of the distance travelled through accelerometer should match with the distance between two consecutive GPS locations. Hence, a sensor-based anti-emulation-detection system should be compliance to all these scenarios so that the use of sensor's value in an innovative way (as described above) cannot reveal the identity of the underlying system.

Challenge *(iii)* is related to the introduction of a new sensor into the system. If a new sensor is included in the system, either it is an independent or dependent sensor (sensor S_{11} as shown in Figure 1), the system should be flexible to reprogram so that new sensors can also be adopted for providing anti-emulation-detection capability.

Algorithm 1: Generate Handle for Sensors

Input : $sensors_{obj}, dependSens_{obj}$ // List of sensors and dependencies objects
Output: $sensors_{hndl}$ // Ordered list of handles to generate realistic sensors values

$sensors_{hndl} \leftarrow \phi$
$Unprocessed_{chld} \leftarrow \phi$ // Sensors queue whose child is not processed
$Processed_{chld} \leftarrow \phi$ // List of sensors whose child is already processed
$Dependecy_{graph} \leftarrow generate\_graph(dependency_{obj}, sensors_{obj})$
$Independent_{sensors} \leftarrow getZeroInDegreeNodes(Dependecy_{graph})$
foreach S *in* $Independent_{sensors}$ **do**
    $S_{hndl} \leftarrow default_{hndl}(sensors_{obj}, S)$
    $append(sensors_{hndl},\ (S,\ S_{hndl}))$
    $append(Unprocessed_{chld},\ S)$
while $\neg(empty(Unprocessed_{chld}))$ **do**
    $S \leftarrow dequeue(Unprocessed_{chld})$
    $childs \leftarrow getChilds(Dependecy_{graph}, S)$
    foreach C *in childs* **do**
        $dep_{func} \leftarrow getDep_{func}(dependency_{obj}, (S, C))$
        $C_{hndl} \leftarrow generate_{hndl}(sensors_{obj}, C, dep_{func})$
        if C *not in* $sensors_{hndl}$ **then**
            $append(sensors_{hndl},\ (C,\ C_{hndl}))$
        else if C *is in* $Processed_{chld}$ **then**// Handling cyclic dependency
            $dep_{func} \leftarrow getDep_{func}(dependency_{obj}, (S, C))$
            $C_{hndl} \leftarrow generate_{hndl}(sensors_{obj}, C, dep_{func})$
            $update_{hndl}(sensors_{hndl},\ (C,\ C_{hndl}))$
        else
            $update_{hndl}(sensors_{hndl},\ (C,\ C_{hndl}))$
        if C *not in* $Unprocessed_{chld}$ *and* C *not in* $Processed_{chld}$ **then**
            $append(Unprocessed_{chld},\ C)$
    $append(Processed_{chld},\ S)$
return $sensors_{hndl}$

To emulate realistic values for sensors, one should consider all the scenarios, as discussed above. Hence, a fine-grained method is needed to emulate sensors reading while maintaining the dependencies between them along with the re-programmable capability to adopt new sensors in the system.

To address all the challenges as mentioned above, we present Algorithm 1, which takes two lists. One list holds the available sensor object ($sensors_{obj}$) and the other is related to the dependency between sensors ($dependSens_{obj}$). A sensor's object comprises of sensor's identity (like accelerometer, GPS), a default handle and the initial value. The default handle is useful when a sensor does not depend on others (independent sensors), and the initial value is used to initialize the sensor. On the other hand, a dependency object comprises the identity of two sensors S_i and S_j, and a dependency function F_{ij}, which represents the dependency between S_i and S_j. These two lists have to be provided by a user, and Algorithm 1 generates an ordered list of sensors handle ($sensors_{hndl}$), which can be executed at the analysis time to emulate the sensor's value while preserving the relationship between them.

In Algorithm 1, $Unprocessed_{chld}$ denotes a queue of sensors whose immediate child needs processing w.r.t. its handle to emulating the sensor value, whereas $Processed_{chld}$ holds the list of sensors whose child has already been processed. Apart from storing processed sensors, the algorithm utilizes this list to break any cyclic dependency (see dependency among sensors S_6 to S_9 in Figure 1), which is

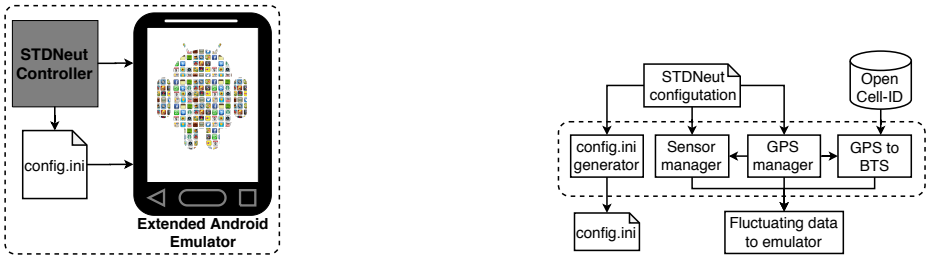
a rare case for sensors. As shown in line 4, the algorithm generates a dependency graph among sensors by using the list of $dependSens_{obj}$ and $sensors_{obj}$. Line 5 gets the list of independent sensors from the dependency graph from where actual learning of sensor handle starts. From lines 6 to 9, the algorithm obtains a handle for each independent sensor, which is equivalent to the default handle in sensor object. The default handle is used to generate the value for a sensor, which does not depend on other sensors. Apart from the sensor handle, independent sensors are then appended in the $Unprocessed_{chld}$ queue, because the children of these sensors may require a handle.

From lines 10 to 26, the algorithm generates the handles for the dependent sensors. The algorithm terminates when the $Unprocessed_{chld}$ queue does not contain any sensor for processing. Line 14 gets the dependency function between parent sensor S and the child sensor C by using the $dependSens_{obj}$ and a handle gets generated at line 15. At line 16, it checks if the sensor is not in the list of $sensors_{hndl}$, algorithm directly adds this handle into $sensors_{hndl}$. In other cases, it updates the already learned handle based on the current dependency and the dependency learned earlier. For updating an already learned handle, there can be two possibilities, one is related to cycle (see cyclic dependency in Figure 1 among sensors S_6 to S_9) and the other is when a sensor depends on more than one sensor (See sensor S_4 in Figure 1). A cyclic dependency is resolved at line 18 in Algorithm 1, where a new dependency function is calculated between parent S and child C. To obtain the new dependency function, we utilize the last value of S (referred to as $\bar{S}$ in line 19) to update the handle of C. At last, when all the children of a sensor S are processed, S is added to the $Processed_{chld}$ at line 26. Finally, the algorithm returns an ordered list of $sensors_{hndl}$, which is then used to emulate the sensor's value at run-time. This algorithm handles the challenge *(i)* and *(ii)*. For challenge *(iii)*, if the user updates the list of sensor objects and dependency objects, then it re-generates the sensor handles for all the sensors, including the new sensors.

4.2 STDNeut Overview

STDNeut system provides robust support for anti-emulation-detection that can be used to design an efficient framework for malware analysis. Figure 2 shows the architecture of STDNeut along with the design of its controller. As shown in Figure 2(a), there are two main subsystems of the STDNeut: (i) Extended Android Emulator and (ii) STDNeut Controller (see Figure 2(b)).

Extended Android emulator: It is responsible for spoofing the information related to sensors, telephony systems, and device data. The STDNeut controller and `config.ini` file govern this spoofing information to the Android emulator. Most of the device-specific information, like IMEI, remains constant during the execution time, while the values for sensors and telephony signal fluctuate over time. During the boot time, the Android emulator reads `config.ini` file and configures a virtual device with device-specific information that is unique to it, while the STDNeut controller handles the fluctuating values at run-time.



(a) STDNeut design overview.

(b) Design of STDNeut controller.

Fig. 2. Architecture of STDNeut, an anti-emulation detection system along with the STDNeut controller.

STDNeut controller: It is responsible for launching an App inside the emulator and feeding essential information for anti-emulation-detection. For example, the controller generates a `config.ini` file that is being used by the Android emulator to configure a virtual device with unique values. The controller also manages the hardware/environment generated events that alter the state of an Android device such as available sensors, telephony signal, and many more. This is achieved by frequently feeding-in realistic sensor data while maintaining the correlation with other sensors (as described in Section 4.1 by utilizing Algorithm 1) and other hardware related events into the emulator. To feed the sensor data and hardware-related events, the controller uses the emulator console APIs [6]. Other than the core features mentioned above, the controller also enables and configures other functionalities which simulate incoming calls/SMSes, manipulates signal strength, and many more. We discuss the extension made to Android emulator in the next section.

4.3 Extensions to the Android Emulator

A smartphone contains multiple sources of information that are either unique to a device and does not change during its life or information may get changed over time due to the operating environment that alters its state. Mostly, a device gets a unique identity from the telephony system that includes IMEI, IMSI, phone number, and many more. To interact with the telephony system, we use AT commands [1]. To provide a unique identity to a virtual device, we intercept the AT command request at the emulator layer for spoofing the response. For example, a smartphone makes "AT+CGSN" and "AT+CIMI" commands to query IMEI and IMSI numbers, respectively. This spoofed information is fed to the AT command by concerning the `config.ini` file. Similarly, in response to AT command, other values are also fed that remain constant but unique to a device. Apart from the `config.ini` file, these values can also be supplied to a virtual device using command line arguments. We use the emulator console to supply realistic data periodically for the hardware/environment events that alter the

Algorithm 2: Path patching for GPS trajectory

Input : $Lat_{src}, Long_{src}, Lat_{dst}, Long_{dst}, nSteps$
Output: $trajectory$

$trajectory \leftarrow \phi$
$LatStep_{max} \leftarrow |Lat_{src} - Lat_{dst}| \ / \ nSteps \times 2$
$LongStep_{max} \leftarrow |Long_{src} - Long_{dst}| \ / \ nSteps \times 2$
$Direct_{lat} \leftarrow +1 \ if \ Lat_{dst} > Lat_{src} \quad else \ -1$ // direction
$Direct_{long} \leftarrow +1 if \ Long_{dst} > Long_{src} \quad else \ -1$
$(lat, long) \leftarrow (Lat_{src}, \ Long_{src})$
$append(trajectory, (lat, long))$
foreach $i \ in \ range(0, \ nSteps)$ **do**
    $lat \leftarrow lat + rnd.uniform(0, \ LatStep_{max}) \times Direct_{lat}$
    $long \leftarrow long + rnd.uniform(0, \ LongStep_{max}) \times Direct_{long}$
    $append(trajectory, \ (lat, \ long))$
return $trajectory$

device state. The Android emulator provides most of the hardware like sensors, GPS, signal strength, and others; the data for them can be fed using emulator console. Android emulator does not provide any interface to change the BTS information with whom a device is currently associated. To provide a realistic GPS location, the information about the BTS associated with the device should collaborate. With this observation, we have added the BTS interface through the emulator console, and the STDNeut controller is supplying the realistic BTS identity.

4.4 STDNeut Controller

The primary responsibility of STDNeut controller is to generate `config.ini` file and feed-in the realistic values for the fluctuating sensors and other hardware events. As shown in Figure 2(b), the STDNeut contains four core components: *(i)* config.ini generator, *(ii)* sensors manager, *(iii)* GPS manager, and *(iv)* GPS to BTS.

config.ini generator: It generates the `config.ini` file to spoof device-specific unique information.

Sensor manager: It manages the device sensors by feeding-in realistic data periodically. To generate the value of sensors, it uses the same handles which are obtained through the Algorithm 1. The sensor manager manages all the sensors and other hardware events except the GPS. However, it gathers the next GPS coordinate to be projected by GPS manager so that the sensors on which GPS depends, can generate appropriate values.

GPS manager: The main reason behind the separate manager for the GPS is the correlation between the current GPS location and the previous location. For example, if the current GPS location is Washington DC, it is impossible for a person to reach New York in five minutes. Hence, a random GPS location alerts an App about the emulated environment. Therefore, a precise method is required to feed GPS location to an emulated environment, and GPS manager provides the same. The GPS manager reads the source and destination geo-location and the travel time from the STDNeut configuration file and generates a route by

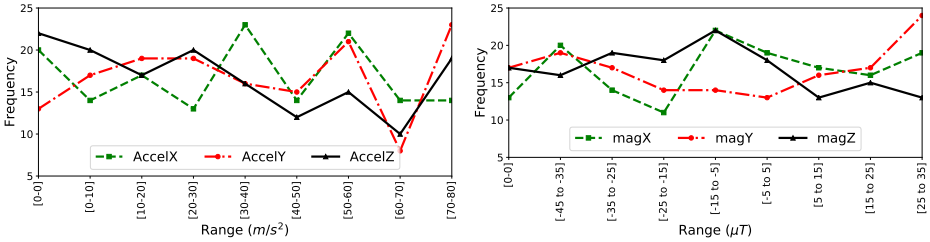
using a path patching algorithm, as shown in Algorithm 2. This algorithm takes source and destination geo-locations along with the number of steps required to move from source to destination, and returns the route trajectory.

GPS to BTS: A realistic GPS location alone is not strong enough to hide an emulated environment. It must be assisted by the BTS location that correlates with the current GPS location. This correlation is based on the maximum distance between the BTS and GPS locations, which may vary from 1 kms to 3 kms depending on the area's population and obstacles. There are several commercial and public services that provide the GPS location by using a BTS ID. Still, no one provides the reverse mapping of it, i.e., providing a BTS ID based on GPS location and the SIM operator that is closer to the current GPS location. GPS to BTS module bridges this gap with the help of the OpenCellID database. The OpenCellID database contains information for the already installed BTS, worldwide, which is publicly available for research purposes. As this database stores BTS information worldwide, an efficient search mechanism is required to retrieve BTS ID based on the current GPS location and SIM operator. With this observation, we first filter the database based on the MCC, followed by the MNC. MCC and MNC reduce the search space to a specific operator within a country. Now we only need location area code and cell-ID to get the desired BTS ID, which is retrieved by calculating the distance with stored BTS location in the database and current GPS location, and compared against the maximum distance allowed. We have used `haversine` formula to measure the distance between the BTS location and current GPS location. The main reason for separate module for GPS to BTS correlation is because it requires to interact with an external database for retrieving the BTS ID according to the GPS location.

5 Validation of STDNeut

We use the Android Open Source Project (AOSP-7.1) to validate the proposed anti-emulation-detection system. For the experiments, Android Virtual Device (AVD) instances were configured with two CPU cores, 1.5 GB of RAM, 2 GB of internal storage and a 512 MB of SD card along with all the sensors.

STDNeut vs. EmuDetLib: We evaluated the effectiveness of the STDNeut against EmuDetLib-Bench and RealMal samples (see Section 3.2). In evaluation, we found that STDNeut remains undetected against all the attacks performed by EmuDetLib-Bench and RealMal samples except the sample under category File info/SysProp and Mix of RealMal. The reason being the use of Qemu specific files and system properties that cannot be spoofed through the emulation layer. Hence, to bypass these detection methods, we have used the Xposed Framework. After evaluating the efficacy of the STDNeut, we attempt to understand this strong defense mechanism's reasoning by performing various experiments. In the remaining part of this section, we discuss the reasons for the efficacy of STDNeut by analyzing different sensor readings and device information during the experiments. We also demonstrate a scenario for understanding the effectiveness of the STDNeut against distributed emulation-detection.



(a) Distribution of accelerometer readings with anti-emulation measures.

(b) Distribution of magneto-meter readings with anti-emulation measures.

Fig. 3. Effectiveness of STDNeut in neutralizing emulation detection using sensors by providing random reading for accelerometer and magnetometer.

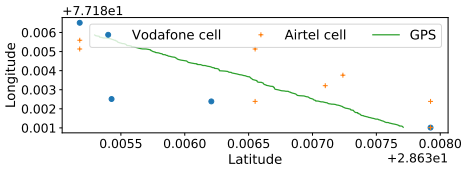


Fig. 4. GPS latitude and longitude reading with anti-emulation measures by feeding-in realistic data along with associated BTS. GPS denotes path trajectory generated using the path patching algorithm.

5.1 Non-detectability through Sensors

To evaluate the efficacy of STDNeut against potential malware exploiting sensor readings, we have developed an App to record and store the values of accelerometer, magnetometer, and GPS readings periodically which are shown in Figure 3 and Figure 4. In this evaluation, we have set two dependencies for sensors, one for time & GPS, and another for GPS & BTS. We make rest of the sensors as independent. The accelerometer reading represents the movement of the device in a three-dimensional space (referred to as AccelX, AccelY, and AccelZ) where the value in each dimension ranges from zero to ninety $(0, 90)$. Figure 3(a) shows the distribution of accelerometer readings where the X-axis represents ranges (total of nine ranges) of sensor values and the Y-axis represents the frequency. We have collected the values by executing an experiment for 150 seconds and reading the sensor values every second. The data shows that all the sensor readings are almost equally likely and approximates a random distribution. Therefore, any emulation-detection technique based on accelerometer reading is nullified by our system. For the magnetometer (Figure 3(b)), the magnetic field readings on each axis in a three-dimensional system are represented as magX, magY and magZ with a range between -45 to $+45$. As shown in the Figure 3(b), the distribution is random, thus it does not allow an emulation-detection scheme using magnetometer data to succeed in detecting the underlying emulation platform.

Another source of emulation-detection is performed by reading GPS data. Unlike accelerometer and magnetometer, GPS data cannot be a random value. Depending on the location of the system, the GPS data should be provided with very slight variations in latitude and longitude. As shown in Figure 4, STDNeut anti-emulation-detection measure can provide valid latitude and longitude values

Table 3. Unique device information provided by STDNeut to three different AVDs executing simultaneously.

Queried Information	Information retrieved		
	AVD1	AVD2	AVD3
PhoneNumber	9876543210	9856543410	9876573213
IMSI	405541385237906	405521385237806	405511385238906
IMEI	359470010002931	359470010302943	359470010002949

along with the associated BTS. In Figure 4, GPS denotes the path trajectory generated using path patching algorithm 2 whereas Vodafone cell and Airtel cell denotes the BTS location in the network of Vodafone and Airtel, respectively.

5.2 Non-detectability through Device Information

Device information is useful in differentiating between an emulated device and a real smartphone. In emulator platforms, device information such as IMEI, IMSI, phone number etc. are either absent or static values are present. To demonstrate the effectiveness of STDNeut's anti-emulation-detection measures, we have used an App called `SIMCardInfo` [15], which extracts the information related to telephony services. We created three instances of this App in three different AVDs and executed all the instances simultaneously for one minute with and without STDNeut. The output of the App queries related to the device information is logged for all instances. We analyzed the log to extract information like IMEI and IMSI. Table 3 shows the captured device information with STDNeut. We are not showing the results other than the proposed system as the device readings were the same for all the instances. As shown in Table 3, STDNeut is capable of providing a unique device identity in a multi-instance setup. This is particularly useful to avoid detection when analyzing potential malware running in separate devices designed to operate in a collaborative manner as all malware see the same device identity. However, the values of PhoneNumber, IMEI, and IMSI are generated manually in the experiment which can be configured through the STDNeut's configuration file without any modification in the Qemu.

5.3 Evading Distributed Emulation-detection

To show the effectiveness of the STDNeut against emulation-detection using multiple clients along with a central server, we used Dendroid [28], a real Android botnet. We integrated EmuDetLib into the Dendroid malware. We modified the Dendroid control server [28] not to send further instructions to the clients that seem to be running on emulated platforms by observing identical device information like IMEI from multiple clients (see Algorithm 2 at `https://skmtr1.github.io/EmuDetLib.html#al2`). Apart from hosting the control server, we also designed a victim site where the malware-infected devices perform a denial of service attack in a distributed manner when instructed from the control server. We created two instances for each of the CuckooDroid and STDNeut, and then executed Dendroid malware with integrated emulation-detection

library. The control server instructs the infected devices to perform an HTTP flood on the victim site mentioned above only if the control server does not detect emulation. In our evaluation, we found that the control server is sending instructions only to the STDNeut system instances and not to the CuckooDroid instances. This was primarily because, the phone number, IMEI, etc. provided to the control server by the CuckooDroid were identical for both the instances, which was not the case with STDNeut. Therefore, we can conclude that the proposed STDNeut system can prevent emulation-detection orchestrated in a distributed setup.

5.4 Discussion and Limitations

Even though STDNeut provides a strong defense against all the malware samples, it falls short in the presence of malware that uses Qemu specific file and system properties for emulation-detection. To overcome this limitation, we have utilized the Xposed framework. The Xposed framework itself is susceptible of detection from App. For example, the Snapchat App uses the native code to detect Xposed [2]. It is possible because Xposed capability is limited to the framework level API only, and here detection is performed through the native code. A more suitable defense is to use kernel-level modification that remains undetected even when the attack is performed from any layer above the kernel. However, our malware set does not contain any samples that detect the existence of Xposed.

Additionally, the first eight digits of IMEI are called TAC (Type Allocation code), which indicate the device type. Malware can also use TAC to detect an emulated environment by observing TAC's mismatch with the Android device name. To our knowledge, we have not observed the existence of such malware. However, STDNeut is a generic solution that requires analyst intervention to configure it with appropriate information like selecting device type and corresponding TAC value in IMEI and other such information.

Furthermore, some Android devices like tablets may lack cellular capabilities or do not have some sensors like GPS. In STDNeut, cellular, GPS, and other sensors fall under the sensor category. An analyst may configure STDNeut without these sensors information to create a realistic emulated device where such sensors are not present.

Evaluation summary: In a nutshell, the proposed STDNeut can effectively execute a malware without being detected as an emulated environment.

6 Conclusion

This paper proposed a flexible and configurable emulation-detection library (EmuDetLib) that provides extensive emulation-detection methods. We have used EmuDetLib to show that anti-emulation-detection measures of the existing dynamic analysis frameworks are not sufficient. Moreover, beyond basic defense against emulation-detection, all the analysis frameworks fail to hide the underlying emulation layer. To design a robust analysis framework on emulated

platforms, we proposed STDNeut, a configurable anti-emulation-detection system. STDNeut hides the emulated platform effectively by handling the data from sensors, telephony system, and device attributes in a realistic manner. We performed experiments to demonstrate the effectiveness of STDNeut against the primary and extended detection methods. We believe STDNeut provides efficient and secure anti-emulation-detection measures that are difficult to be bypassed even by sophisticated malware.

Acknowledgements. We thank our shepherd Matthias Whlisch and all the anonymous reviewers for their helpful comments and suggestions. This work is supported by Visvesvaraya Ph.D. Fellowship grant MEITY-PHD-999.

References

1. AT Commands - 3GPP TS 27.007 (2020), https://doc.qt.io/archives/qtextended4.4/atcommands.html
2. AeonLucid: Snapchat detection on Android Aeonlucid (2019), https://aeonlucid.com/Snapchat-detection-on-Android/
3. AG, G.S.: A new malware every 7 seconds (2019), https://www.gdatasoftware.com/news/2018/07/30950-a-new-malware-every-7-seconds
4. Allix et al.: Androzoo: Collecting millions of android apps for the research community. In: MSR. pp. 468–471 (2016)
5. Android Developers: Run apps on the android emulator — android developers (2019), https://developer.android.com/studio/run/emulator
6. Android Developers: Send emulator console commands — Android developers (2019), https://developer.android.com/studio/run/emulator-console
7. Arakawa, Y.: Emulatordetector: Android emulator detector unity compatible (2019), https://github.com/mofneko/EmulatorDetector
8. Arzt et al.: Droidbench 3.0 (2019), https://github.com/secure-software-engineering/DroidBench/tree/develop
9. Bellard, F.: Qemu, a fast and portable dynamic translator. In: ATEC. p. 41 (2005)
10. Costamagna et al.: Identifying and evading android sandbox through usage-profile based fingerprints. In: RESEC (2018)
11. Desnos et al.: Welcome to Androguard's documentation! - androguard 3.3.5 documentation (2019), https://androguard.readthedocs.io/en/latest/
12. Diao et al.: Evading android runtime analysis through detecting programmed interactions. In: WiSec. pp. 159–164 (2016)
13. Fenton, C.: Android emulator detect — calebfento (2019), https://github.com/CalebFenton/AndroidEmulatorDetect
14. Gingo: Android-emulator-detector: Small utility for detecting if your app is running on emulator, or real device (2019), https://github.com/gingo/android-emulator-detector
15. Gonzalez, H.: Sim card info - apps on google play (2019), https://play.google.com/store/apps/details?id=me.harrygonzalez.simcardinfo&hl=en_IN
16. IDC: IDC–smartphone market share - OS (2019), https://www.idc.com/promo/smartphone-market-share/os
17. Inc., F.: Android emulator detector: Easy to detect android emulator (2019), https://github.com/framgia/android-emulator-detector

18. Jing et al.: Morpheus: Automatically generating heuristics to detect android emulators. In: ACSAC. pp. 216–225 (2014)
19. Kudrenko, D.: Emulator-detector: Detect emulators like genymotion and nox player by accelerometer (2019), https://github.com/dmitrikudrenko/Emulator-Detector
20. Lab, A.: Argus SAF - argus-pag (2019), http://pag.arguslab.org/argus-saf
21. Lantz, P.: An Android Application Sandbox for Dynamic Analysis. Master's thesis (Nov 2011), https://www.eit.lth.se/sprapport.php?uid=595
22. Lockheimer, H.: Android and security - official google mobile blog (2012), http://googlemobile.blogspot.com/2012/02/android-and-security.html
23. Maruyama et al.: Base transceiver station for w-cdma system. Fujitsu scientific & technical journal **38**, 167–173 (01 2002)
24. MobSF Team: 1. documentation . MobSF/mobile-security-framework-MobSF wiki (2019), https://github.com/MobSF/Mobile-Security-Framework-MobSF/wiki/1.-Documentation
25. Oberheide, J., Miller, C.: Dissecting the android bouncer (2012), https://jon.oberheide.org/files/summercon12-bouncer.pdf
26. Orlowski, A.: Google play store spews malware onto 9 million 'Droids. the register (2019), https://www.theregister.co.uk/2019/01/09/google_play_store_malware_onto_9m_droids/
27. Percoco, N.J., Schulte, S.: Adventures in BouncerLand (2012), https://media.blackhat.com/bh-us-12/Briefings/Percoco/BH_US_12_Percoco_Adventures_in_Bouncerland_WP.pdf
28. qqshow: Github - qqshow/dendroid: Dendroid source code. contains panel and apk. (2019), https://github.com/qqshow/dendroid
29. Rasthofer et al.: Harvesting runtime values in android applications that feature anti-analysis techniques. In: NDSS (2016)
30. Sadeghi et al.: A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software. IEEE Transactions on Software Engineering **43**(6), 492–530 (June 2017)
31. Sun et al.: TaintART: a practical multi-level information-flow tracking system for android runtime. In: ACM SIGSAC CCS. pp. 331–342 (2016)
32. Tam et al.: Copperdroid: Automatic reconstruction of android malware behaviors. In: NDSS (2015)
33. Tam et al.: The evolution of android malware and android analysis techniques. ACM Comput. Surv. **49**(4), 76:1–76:41 (Jan 2017)
34. Technologies, C.S.: CuckooDroid book (2014), https://cuckoo-droid.readthedocs.io/en/latest/
35. thehackernews.com: New android malware apps use motion sensor to evade detection (2019), https://thehackernews.com/2019/01/android-malware-play-store.html
36. Vidas, T., Christin, N.: Evading Android runtime analysis via sandbox detection. In: ASIA CCS (2014)
37. Wang et al.: Droid-AntiRM: taming control flow anti-analysis to support automated dynamic analysis of android malware. In: ACSAC (2017)
38. Wei et al.: Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In: ACM SIGSAC CCS (2014)
39. XDA Developers: Xposed framework hub (2019), https://www.xda-developers.com/xposed-framework-hub/
40. Yan, L.K., Yin, H.: Droidscope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis. In: USENIX Security (2012)