Published: 31 March 2023
Online AM: 12 October 2022
Accepted: 22 September 2022
Revised: 08 September 2022
Received: 11 February 2022

RESEARCH-ARTICLE

InviSeal: A Stealthy Dynamic Analysis Framework for Android Systems

SAURABH KUMAR, Indian Institute of Technology Kanpur, Kanpur, UP, India

Cyber Security, Mobile Security, Malware Analysis

DEBADATTA MISHRA, Indian Institute of Technology Kanpur, Kanpur, UP, India

BISWABANDAN PANDA, Indian Institute of Technology Bombay, Mumbai, MH, India

SANDEEP KUMAR SHUKLA, Indian Institute of Technology Kanpur, Kanpur, UP, India

InviSeal: A Stealthy Dynamic Analysis Framework for Android Systems

SAURABH KUMAR and DEBADATTA MISHRA, Indian Institute of Technology Kanpur, India
BISWABANDAN PANDA, Indian Institute of Technology Bombay, India
SANDEEP KUMAR SHUKLA, Indian Institute of Technology Kanpur, India

With wide adaptation of open-source Android into mobile devices by different device vendors, sophisticated malware are developed to exploit security vulnerabilities. As comprehensive security analysis on physical devices are impractical and costly, emulator-driven security analysis has gained popularity in recent times. Existing dynamic analysis frameworks suffer from two major issues: (i) they do not provide foolproof anti-emulation-detection measures even for fingerprint-based attacks, and (ii) they lack efficient cross-layer profiling capabilities. In this work, we present InviSeal, a comprehensive and scalable dynamic analysis framework that includes low-overhead cross-layer profiling techniques and detailed anti-emulation-detection measures along with the basic emulation features. While providing an emulator-based comprehensive analysis platform, InviSeal strives to remain behind-the-scene to avoid emulation-detection. We empirically demonstrate that the proposed OS layer profiling utility to achieve cross-layer profiling is ~1.26× faster than existing `strace`-based approaches. Overall, on average, InviSeal incurs ~1.04× profiling overhead in terms of the number of operations performed by the various workloads of the CaffeineMark-3.0 benchmark, which is better than the contemporary techniques. Furthermore, we measure the anti-emulation-detection strategies of InviSeal against the fingerprint-based emulation-detection attacks. Experimental results show that the emulation-detection attacks carried out by the malware samples do not find InviSeal as an emulated platform.

CCS Concepts: • **Security and privacy → Malware and its mitigation**; **Mobile platform security;**

Additional Key Words and Phrases: Android, malware analysis, sandbox, anti-emulation-detection, memory forensics

ACM Reference format:
Saurabh Kumar, Debadatta Mishra, Biswabandan Panda, and Sandeep Kumar Shukla. 2023. InviSeal: A Stealthy Dynamic Analysis Framework for Android Systems. *Digit. Threat.: Res. Pract.* 4, 1, Article 13 (March 2023), 31 pages.
https://doi.org/10.1145/3567599

1 INTRODUCTION

Android [20] is common in modern-day mobile devices because of its open-source availability and robust support for mobile application development. According to a recent report published by **International Data Corporation (IDC)**, the global market share of Android OS was 84.1% [4] in the third quarter of the year 2020. Further, the simplicity of the Android framework with regards to the application development has resulted in substantial

This work is partially supported by Visvesvaraya Ph.D. Fellowship Grant No. MEITY-PHD-999, SERB, and DST through C3i center and C3i hub projects at IIT Kanpur.
Authors' addresses: S. Kumar, D. Mishra, and S. K. Shukla, Indian Institute of Technology Kanpur, Kanpur, Uttar Pradesh, India, 208016; emails: {skmtr, deba, sandeeps}@cse.iitk.ac.in; B. Panda, Indian Institute of Technology Bombay, Mumbai, Maharshtra, India, 400076; email: biswa@cse.iitb.ac.in.

2576-5337/2023/03-ART13 $15.00
https://doi.org/10.1145/3567599

growth in number of mobile applications developed world wide. A study from Statista shows that every day more than 3.5K Android applications were released in the year of 2020 [5]. As a consequence of large scale adaptation of Android OS and ever increasing contributions in the Android application space, security has become a non-trivial challenge recently. A study related to malware activities in Android platforms published by GDATA shows that in the first half of 2018, more than 2 million new Android malware were recorded [16]. In other words, a new Android malware is *born every seven seconds* [16].

Existing approaches to 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 [51, 61]. Techniques based on only static analysis [21, 41, 49, 50, 52, 71] 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 [13], Java-reflection mechanisms, and code encryption to bypass the static analysis-based detection [60, 66]. Dynamic analysis and detection techniques are the most commonly used and researched in the contemporary mobile security arena and is the scope of this article.

Existing dynamic analysis and detection techniques address specific security issues like information leakage through device channels, untrusted mobile activities through different applications in a secure environment. For example, **dynamic information flow tracking (DIFT)**-based malware detection techniques like taintDroid [40] and taintART [64] track the information flow from the source (sensitive information) to sink (device channel) to detect information leakage while behavior graph-based techniques [63] identify potential malware based on call graphs. Other techniques are based on network traffic monitoring [31] and cloud usage monitoring [44] to detect potential malware. However, all of the above techniques are limited to specific security risks and do not provide a comprehensive platform for malware analysis and detection. For example, taintART [64] provides information tracking only at the framework layer that can not detect malware leaking information through native code. Moreover, techniques like taintART when used on a real device *results in significant performance overheads (~10%) and increased energy usage.*

Generic sandbox-based techniques (Droidbox [53], CuckooDroid [67], DroidScope [77], MobSF [58]) provide a sandboxed virtual environment to perform malware analysis and detection by executing applications inside the sandboxes and collecting various event logs.

The problem: Even though a wide variety of Android sandboxes are available for application analysis, malware can bypass the dynamic analysis process running on these frameworks by employing one or more techniques listed below.

(i) Many malware [70] 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 methods adopted by malware. For example, CuckooDroid [67] provides fixed data when different device-specific information (e.g., GPS and IMEI) is queried from the application, which allows a malware to observe them and evade the dynamic analysis defenses [62].

(ii) CuckooDroid and taintART-based sandboxes depend only on the profiling information collected from the framework layer, therefore *can not detect information leakage through the native code and the OS level system call APIs.* For example, two colluding malicious applications can use system calls (like `mmap()`, socket, file operations) to setup a communication channel without being detected by the above techniques.

(iii) `Strace` [7] is a widely used utility to profile system call information to enable cross-layer profiling. However, `strace` incurs high overhead and slows down the application execution, opening another way to detect the monitoring environment. Also, `strace`-based profiling system can be detected from the malware using a simple detection method where the malware launches `strace` on itself. It is possible because an application can employ strace on itself without superuser permission. In this scenario, given the `strace`-based dynamic analysis already

tracing the malware application, the `strace` instance launched by the malware will fail, which exposes the dynamic analysis process.

Our goal: Ideally a desirable mobile emulator platform for security analysis should provide the following features: *(i)* it must have cross-layer (application level to OS level) profiling capabilities, *(ii)* built-in anti-emulation-detection measures for robust malware analysis, and *(iii)* incur low profiling overheads. Apart from the above mentioned features, memory dump and packet capture features should also be supported, which may be used if required for offline analysis.

Our proposal: In this article, we present InviSeal, a stealthy, low overhead, and comprehensive dynamic analysis framework for applications in the ART execution environment. First, we develop **Sensor, Telephony system, and Device state information Neutralizer (STDNeut)** to bypass the sensors-based emulation-detection strategy that is fully designed using Qemu [25]-based Android emulator [19]. Second, we design ARTmon using `Droidmon` module (based on `Xposed` [37] framework) to bypass file and system properties-based emulation-detection along with the ability to instrument framework level API. Furthermore, we develop an OS-level system call interposition technique to profile the system call activities in an efficient manner. We provide a one-place log storage system for further analysis and detection of malware. Overall, our contributions are as follows,

- We empirically evaluate well-known existing dynamic analysis frameworks against the basic and extended emulation-detection attacks of EmuDetLib [48] and the real malware samples of the year 2019 (Section 3.1). The evaluation shows that existing frameworks do not have a foolproof way to stop emulation-detection attacks that rely on fingerprinting.
- We design STDNeut by using the insights of the empirical validation of existing frameworks 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 information (Section 4). As part of STDNeut, we propose an algorithm to generate realistic data for sensors while addressing the associated challenges (Section 4.1).
- We modify the `Droidmon` module to support file and system properties-based anti-emulation-detection measures (referred to as ARTmon, Section 5). We develop SysCallMon (Section 6), a configurable low-overhead OS-level utility (kernel module) to monitor system calls invoked from applications. However, the default kernel provided by the Android SDK does not allow kernel functionality extension through the kernel module. Therefore, the Android emulator kernel needs to be recompiled by turning on the support for loadable kernel modules so that SysCallMon can work seamlessly.
- We propose a comprehensive Android security audit framework based on the above utilities that provides a single-point dynamic profiling and analysis support (Section 7).

To evaluate the efficacy of InviSeal, we perform several experiments. Summary of our evaluations are as follows:

- Our evaluation of InviSeal shows that on average the profiling overheads is ~1.04×, which is better than contemporary techniques. Moreover, SysCallMon is ~1.26× faster than `strace`-based system call profiling.
- We measure the anti-emulation-detection strategies of InviSeal against the emulation-detection library and real malware samples for the year 2019 (Section 8.2). Tables 2 and 3 show that the attack carried out by the emulation-detection library and real malware samples do not find InviSeal as an emulated platform. Later, we do a number of other experiments to figure out why InviSeal's defense system works so well (Sections 8.2.1 and 8.2.2)
- Last, we show how InviSeal can be used in practice (Section 8.3). For example, the benefit of cross-layer profiling to find collusion attacks (Section 8.3.1), distributed emulation-detection can be avoided (Section 8.3.2), and the memory forensics approach to extract dynamically loaded code (Section 8.3.3).

The rest of the article is organized as follows. Section 2 discusses the background of the Android OS and the `Xposed` hooking framework. Section 3 presents the empirical evaluation of existing frameworks that serve as

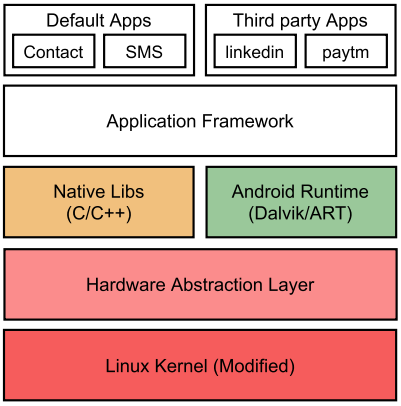


Fig. 1. Android platform architecture [35].

motivation and defines the threat model. We discuss the design and implementation details of STDNeut, ARTmon, and SysCallMon in Sections 4, 5, and 6, respectively. We build the InviSeal in Section 7, whereas we present the evaluation of it in Section 8. Section 9 presents the related work. We discuss more details about the InviSeal and future enhancements in Section 10 before concluding the work in Section 11.

2 BACKGROUND

In this section, we discuss the Android platform architecture and explain the working of `Xposed` framework and `Droidmon` module. After that, we provide an overview of the **Base Transceiver Station (BTS)** as a smartphone frequently communicates with it.

2.1 Android Platform

Android is an open-source, Linux-based software stack created for a wide array of mobile devices. The major components of the Android platform are shown in Figure 1. It comprises six components: the Linux Kernel, hardware abstraction layer, native libraries, Android runtime, application framework (Java API Framework), and an application Layer (default applications or third party applications) [35]. Working of each component is as follows:

Linux Kernel: The foundation of the Android Platform is the Linux OS (a.k.a. Linux Kernel). The **Android virtual machine (ART/DVM)** depends on the Linux kernel for underlying functionalities such as memory management, threading, power management, and so on. Android takes advantage of many key security features provided by Linux such as process level isolation, user-group-based file permissions for privilege separation, and so on.

Hardware Abstraction Layer (HAL): The HAL provides standard interfaces exposing device hardware capabilities to the higher-level Application Framework. The HAL layer is realized through Linux kernel system call APIs. When a framework API makes a call to access the device hardware for the first time, the Android system loads the library module for that hardware component.

Native Libraries: Many core components of Android systems and services such as ART and access to HAL APIs require invoking methods provided by native libraries written in C/C++. The Android platform also provides Java framework APIs to expose the functionality of some of these native libraries to the applications. Some of these Java framework APIs include media server framework, SQLite Database, Libs (bionic), and so on.

Android Runtime: It is the application runtime of the Android, a virtual machine similar to JVM. The difference between ART and JVM is as follows, JVM is a stack-based virtual machine, whereas ART is register-based.

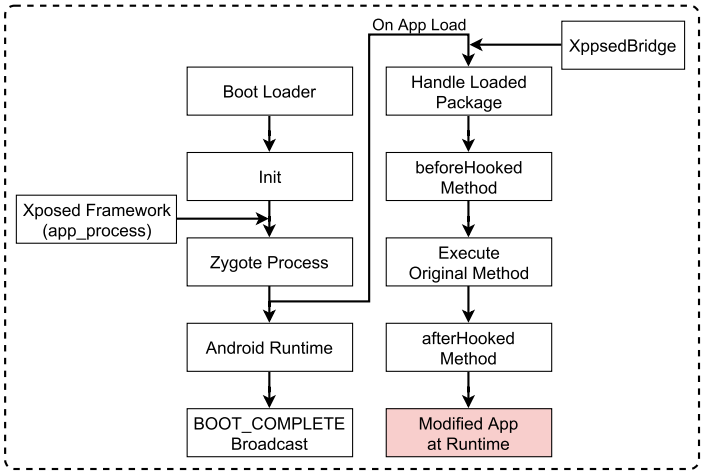


Fig. 2. Work flow of Xposed hook framework.

Therefore, ART incurs less performance overhead as compared to the JVM. Android uses two types of virtual machines: Dalvik virtual machine (Android version < 5) and ART (recent Android releases).

Application Framework: It provides the Java API to facilitate application development in the Android mobile platform. For example, application framework provides APIs to use view system objects like button, text view, list control, and so on. It also provides APIs to access information from resource manager, activity manager, and so on.

Application Layer: This layer contains actual Android applications. There are two types of applications available with Android: system (default) applications and third party applications. System applications are developed by device vendor and shipped with the device (e.g., contacts, browser). However, third party applications are the custom build applications and hosted on the market (Google play) from where a device user can install depending on her requirements (e.g., Whatsapp and Uber). Note that, system applications have higher privileges compared to the third party applications.

2.2 Xposed Framework

`Xposed` [37] is a widely used generic hooking framework for the Android platform. APIs provided by the Xposed framework allow us to modify an application's behavior by injecting our code before and after any Java methods or by replacing individual ones. To work appropriately, `Xposed` requires a rooted device. It has mainly four components: `Xposed`, `XposedBridge`, `XposedInstaller` [54], and `XposedMods`. `Xposed` and `XposedBridge` provides support to accommodate the hook framework inside an application. `XposedInstaller's` main responsibility is to manage the `Xposed` framework and modules developed by other developers on top of `Xposed`. `XposedMods` are the modules designed to perform a specific task such as changing the behavior of an existing device. Figure 2 illustrates the overall workflow of the `Xposed` framework.

When an Android phone starts, the boot-loader calls the kernel, which loads the first process called `init` at the end of OS boot. The `init` process is responsible for setting up all the daemon processes and components of the Android platform. After the successful start of the `init` process, it invokes the core Android component called Zygote, which is responsible for loading other applications. In an `Xposed` hooked system, a modified Zygote process is loaded by the `Xposed` framework to hijack the overall control of the system. After this step, whenever any application is executed, the modified Zygote loads the application by setting up the execution environment

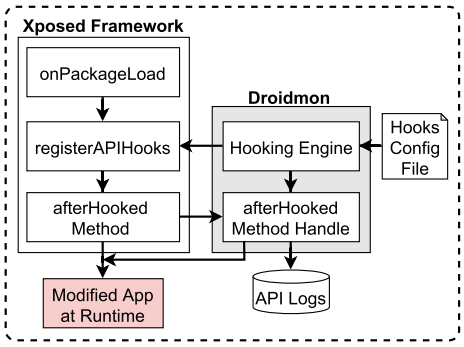


Fig. 3. Droidmon design using Xposed framework hooks.

for them. At the beginning of the application execution, `Xposed` loads the `XposedBridge` library as part of the executable. The `XposedBridge` library provides the necessary APIs used by the `Xposed` module to change the application behavior. An `Xposed` module registers the API to be hooked with the `Xposed` framework along with the handler methods and invokes it before and after executing a hooked API. As a result, the `Xposed` module can be used to change the behavior of any application at run time.

Other `Xposed` like hook-based frameworks such as ARTist [24] and ARTDroid [33] are also proposed. However, these frameworks are not as popular as `Xposed` as they lack necessary software support.

2.2.1 Droidmon. `Droidmon` [45] is an `Xposed` module designed for monitoring the framework level APIs used by applications in CuckooDroid [67] in a configurable manner. Figure 3 shows the design of `Droidmon` module leveraging the hook support provided by `Xposed` framework. When an application package loads, hook engine of `Droidmon` parses the hooks configuration file and provides the details of the API to the `Xposed` framework for instrumentation. The `Xposed` framework (explained before) enables hooks at different application execution points. `Droidmon` primarily makes use of the `afterHooked` method (refer Figure 2) to record information related to the invoked API (e.g., return value and arguments). `Droidmon` is used to monitor the API in a non-intrusive manner without modifying the original application methods during the monitoring phase.

2.3 Base Transceiver Station (BTS)

BTS [57] 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 and is also a source of geo-location of a device. 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 [72]. Malware can use BTS-provided geo-location and check it with the GPS-provided geo-location to detect the emulated platform. Therefore, the BTS location must correlate with the GPS location to nullify the location-based emulation-detection attack, which is generally missing in the emulated platform.

3 MOTIVATION

3.1 Anti-Emulation-Detection Measures

Virtual environment-based dynamic analysis techniques described before can provide effective detection against potential malware. However, smart malware developers can evade dynamic analysis in the virtual environment

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.

Table 2. Defense Mechanisms Provided by Existing Dynamic Analysis Tools Against Different Types of Emulation-Detection Methods of EmuDetLib

Detection Type	Sub-type	Emulator	Droidbox	CuckooDroid	MobSF	DroidScope	taintART	AMS	InviSeal
Unique Device Information	Basic	×	✓	✓	✓	✓	✓	✓	✓
	Smart	×	×	×	×	×	×	✓	✓
Sensors	–	×	×	×	×	×	×	×	✓
Device State	–	×	×	×	×	×	×	×	✓
GPS	–	×	×	×	×	×	×	×	✓
Distributed	–	×	×	×	×	×	×	✓	✓

Note: ✓ represents successful in bypassing the emulation-detection attack by underline emulated, whereas × represents failure in bypassing the emulation-detection attack. We use this notation in the rest of the tables. AMS stands for Android Malware Sandbox.

by detecting the emulator through tricks like delusion [75], red pill [59], and fingerprint [26, 46, 70]. Malware that hides by using fingerprints is common today, and it is easy to make so that it can bypass the analysis process running on emulators. Therefore, dynamic analysis tools must provide anti-emulation-detection defense mechanisms to out-smart the fingerprint-based emulation-detection techniques employed by the malware. Malware can detect emulated devices by employing one or more of the following emulation-detection techniques (see Table 1, for more details, please refer to emulation-detection library EmuDetLib [48]), which are based on fingerprint: *(i)* **Unique device information (UDI)**, *(ii)* Sensors reading, *(iii)* Device state information, *(iv)* GPS information, and *(v)* Distributed detection.

In an ideal dynamic analysis framework, malware should not be able to detect the underlying emulated environment. To understand the defense measures opted by existing dynamic analysis tools along with the vanilla Android emulator (referred to as emulator), we evaluate them against the EmuDetLib [48]. Apart from the EmuDetLib-Bench, we have collected 1,000 malware from the AndroZoo [17] project where the dex date is of the year 2019 and also downloaded motion sensor malware disclosed by the TrendMicro [68] (referred to as RealMal) to evaluate the existing frameworks. We have considered CuckooDroid [67], Droidbox [53], MobSF [58], DroidScope [77], taintART [64], **Android Malware Sandbox (AMS)** [22] along with the vanilla Android emulator (referred to as emulator) [19] as the candidate analysis systems for the empirical study, as they are readily available. We exclude the online analysis systems and other sandboxes (no longer available) 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.

Table 2 shows the evaluation result of the emulation-detection of the candidate sandboxes against all the detection methods available in EmuDetLib. While all of the existing techniques can provide defense against emulation detection by stand-alone malware expecting non-random device values (basic unique device information) except for vanilla emulator, they fall short in addressing other emulation-detection methods. For example, `Droidbox`

Table 3. Evaluation of Existing Framework Against Real Malware Sample

Detection Type	#Sample	Emulator	Droidbox	CuckooDroid	MobSF	DroidScope	taintART	AMS	InviSeal
No emulation-detection	284	✓	✓	✓	✓	✓	✓	✓	✓
UDI	137	×	✓	✓	✓	✓	✓	✓	✓
File Info / SysProp	303	×	✓	✓	✓	✓	✓	✓	✓
Device State	19	×	×	×	×	×	×	×	✓
Sensors	3	×	×	×	×	×	×	×	✓
Mix	257	×	✓	✓	✓	✓	✓	✓	✓

Note: ✓ represents successful in bypassing the emulation-detection attack by underline emulated platform, whereas × represents failure in bypassing the emulation-detection attack. UDI represents Unique Device Information. SysProp represents the system properties. The Mix represents the malware sample that uses more than one detection method from Unique Device Information, File Info and System Properties.

provides IMEI value, which looks genuine and can fool some of the malware designed to work in a stand-alone setup. However, with their default setting, existing tools and techniques fail to provide anti-emulation-detection defense in all other cases (as listed in Table 1). Only AMS is the tool that provides flexibility to change the settings to thwart the smart UDI and distributed emulation-detection attack by modifying UDI values in the corresponding scripts. It is possible because AMS uses the FЯIDA [10] framework that exposes the Javascript APIs to change the behavior of an application. Still, AMS fails to defend against emulation-detection attacks using sensors and device state information.

```
1  public void onSensorChanged(SensorEvent paramSensorEvent) {
2    this.k.registerListener(this, this.l, 3);
3    Sensor sensor = paramSensorEvent.sensor;
4    this.k.registerListener(this, sensor, 3);
5    if (sensor.getType() == 1) {
6      float[] arrayOfFloat = paramSensorEvent.values;
7      float f1 = arrayOfFloat[0];
8      float f2 = arrayOfFloat[1];
9      float f3 = arrayOfFloat[2];
10     long l = System.currentTimeMillis();
11     if (l - this.m > 100L) {
12       long l1 = this.m;
13       this.m = l;
14       if (Math.abs(f1 + f2 + f3 - this.n - this.o - this.p) / (float)(l - l1) * 10000.0F > 600.0F)
15         a();
16       this.n = f1;
17       this.o = f2;
18       this.p = f3;
19     }
20   }
21 }
```

Listing 1. Code snippet from motion sensor malware.

Similarly, on executing samples of RealMal (see Table 3), 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 the malware sample (RealMal) under sensors. Listing 1 shows the code snippet from BatterySaverMobi (a real malware), 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.

Our proposal (STDNeut, Section 4 and ARTmon, Section 5) bridges this gap by employing intelligent anti-emulation-detection measures in the emulation platform.

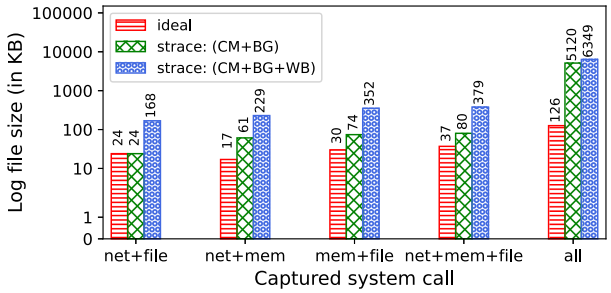


Fig. 4. Storage overhead of strace-based system call logging w.r.t. ideal (targeted) logging using CaffeineMark (CM) along with background applications (BG) and a web browser (WB). Lower is better.

3.2 System Call Monitoring

Tracking application interaction with the OS through system calls can provide useful insights regarding the application behavior. Strace [7] is a well-known utility available in all Linux-based operating systems, including Android OS. A typical usage of strace is to execute an application using strace to capture all the system calls used by the application from the beginning. However, this is not the case with the Android OS due to the following reason.

Unlike application on traditional computer systems where application entry point is fixed (e.g., the main() function), an Android application has multiple entry points. Execution of Android application starts from one of the multiple entry points by sending an intent (a.k.a. the message passing technique in Android) from the package manager. In such a scenario, strace is unable to capture the system calls used by an application from its start.

To capture system calls invoked by an application from the beginning of its execution, one can apply strace on the Zygote process, which is responsible for setting up the runtime environment for the applications and create new processes. When strace is applied on the Zygote process, it not only captures the system calls made by applications of interest but also the system calls made by other applications not of interest. System-wide system call logging results in a lot of overheads, both in terms of storage and processing power. Moreover, all system calls are not useful for malware detection, so researchers are more interested in profiling a set of system calls for further analysis. Eventhough strace is capable of profiling a selected set of system calls, it also profiles the same set of system calls made by other applications. This results in unwanted system call in the log file and requires parsing to extract application-specific system calls, which can be a lot of overheads and efforts. To show the overheads of strace when used in Android platform, we have performed the following experiment.

We executed Java-based micro benchmark CaffeineMark [32] and captured system call information using strace. We have executed CaffeineMark with two scenarios: *(i)* CaffeineMark with background processes and *(ii)* CaffeineMark with a browser application along with other background processes. We repeated the experiments to capture combinations of different categories of system calls in the above scenarios and compared the capture log size with the ideal capture log size. The ideal capture log size represents the size of the log file when the system call tracing is performed in a targeted manner (similar to proposed solution). Figure 4 compares the ideal log file size with size of the log file generated by strace in different setups. When all system calls are profiled, the ideal log file size is ~40× and ~50× smaller than the log generated by strace when CaffeineMark is executed along with background processes and with browser application, respectively. Even when a subset of system calls are captured with only background processes, strace results in up to 3.5× increased log file size (Figure 4) compared to ideal profiling. Therefore, a sophisticated and low overhead system call profiling utility can improve the efficiency of dynamic analysis in emulated platforms.

Apart from the logging overheads associated with `strace`-based technique, there are some other challenges to analyze logs generated by `strace`. In `strace`, process ID is the only available filter that can be directly applied to parse the log files. However, the process ID cannot be mapped to a specific application from the log file itself and requires support from other utilities like `ps`, `top`, and so on. Further, when `strace` is used to follow child processes (which is the case with Zygote), there are incomplete log entries because of overlapping system calls from multiple processes.

Additionally, employing `strace` to capture the system calls made by an application (as discussed in Section 3.2) slowdowns the system by 1.36× (see Section 8.1). The slowdown introduced by the `strace` opens another way to detect the analysis environment. The primary cause of system slowdown is the underlying `ptrace` system call that `strace` uses to monitor each system call of interest.

Last, a malware can easily bypass the `strace`-based analysis process by launching `strace` on itself. In this case, given the `strace`-based dynamic analysis already tracing the malware, the `strace` launched by the malware will fail, which exposes the dynamic analysis process.

To address these issues related to system call profiling, as part of InviSeal, we propose SysCallMon (Section 6), a Linux kernel extension (kernel module), for low overhead targeted system call profiling.

3.3 Memory Forensics

Memory forensic plays an essential role in finding evidence from the volatile memory in investigating Cyber Crime. Nowadays, malware authors use advanced techniques such as dynamic code loading, code packing, and so on, to bypass static analysis. In the dynamic code loading method, malware loads the code at runtime, which we can obtain from the volatile memory. Similarly, a packed malware first requires the unpacking of packed binary to perform a malicious action. This unpacked code and decryption keys are available in the memory during the execution of malware. The memory forensics approach can be useful in such a scenario to retrieve the evidence (unpacked code, dynamically loaded code) from the volatile memory, and further analysis can be performed on such code to capture the behavior of malware. Keeping this in mind, the researcher has started working on the memory forensics-based malware analysis [56]. In Reference [69], authors have presented the effectiveness of memory forensics in the malware analysis process. Hence, a dynamic analysis tool must support an on-demand functionality to capture volatile memory for further analysis.

3.4 Threat Model

Most dynamic analysis frameworks are built on top of the emulated platform, because it offers fine-grained control over the execution of a given application. However, Malware authors are aware of the emulated platform and try to find it using tricks like delusion [75], red pill [59], and fingerprint (see Section 3.1) to hide their evil intentions and keep them from being caught. Malware that hides by using fingerprints is common today, and it is easy to make so that it can bypass the analysis process running on emulators. Moreover, most dynamic analysis frameworks either focus on a higher level of information to detect malware or use methods that incur a very high overhead on profiling an application that opens up another way to bypass the analysis process. Therefore, we aim to bridge this gap by designing a stealthy dynamic analysis framework that can profile an application across multiple layers while introducing very low profiling overhead. In addition, we assume that emulated platforms are not prone to delusion and red pill attacks. Also, malware does not attempt to detect hooking frameworks like Xposed and FЯIDA [10] to bypass the analysis process.

3.5 Why an Integrated Solution Is Required?

Malware analysts can perform multiple types of analysis ranging from cross-layer profiling to memory forensics to identify malicious behavior of an application while hiding the underlying emulated environment. However, none of the solutions that we are proposing (i.e., STDNeut, ARTMon, and SysCallMon) can provide all

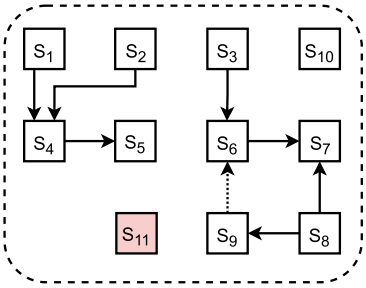


Fig. 5. An example of sensor's dependency graph. Sensor S_{11} in shaded box represents a new sensor introduced in the system.

functionality in one place. For example, STDNeut alone cannot provide framework-level profiling information. It only protects the emulated environment from being detected as an analysis platform by using sensors, telephony systems, or device state information. Similarly, ARTmon alone cannot be used for profiling such malware that includes malicious behavior inside the native code. Moreover, SysCallMon alone neither protects emulated platforms from being detected as an analysis environment nor profile framework-level APIs. Furthermore, none of these solutions provides the ability to perform malware analysis based on memory forensics techniques. Hence, an integrated solution is required that can be used in many malware analysis scenarios without being detected as an emulated platform.

4 STDNeut: DESIGN AND 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. Malware can use these sensors to detect an emulated environment. A recent example of sensor-based detection is the observation of TrendMicro, where malware (in Play Store) makes use of the motion-detection feature to evade the dynamic analysis [68]. We have collected three motion sensor malware and evaluated the existing frameworks against them. The evaluation (see sensors evaluation result in Table 3) shows that non of the existing frameworks provides the defense against the sensors-based emulation-detection attack. Therefore, to nullify the effect of sensors-based emulation-detection, we have identified three main challenges as follows:

(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 5 that represents 11 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 5, some sensors do not depend on other sensors (sensor with zero indegree in directed graph); we name them as independent sensors, whereas sensors with indegree ≥ 1 are called dependent sensors, because the value of these sensors depends on the value of others.

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 $sensors_{hndl}$

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 5, 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 5, 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, then a person cannot reach New York in 5 min. 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 5), the system should be flexible to reprogram so that new sensors can also be adopted for providing anti-emulation-detection capability.

To emulate realistic values for sensors, one should consider all the challenges, 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 list 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. However, 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.

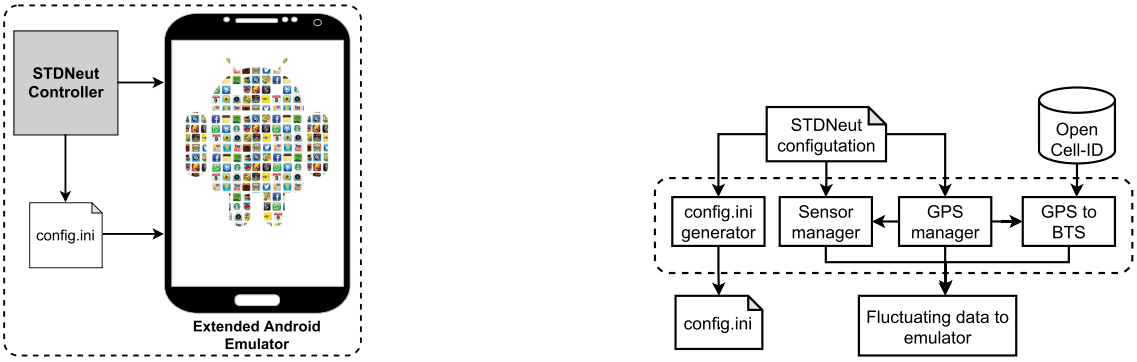
In Algorithm 1, $Unprocessed_{chld}$ denotes a queue of sensors whose immediate child needs processing w.r.t. its handle to emulating the sensor value, whereas $Processed_{chld}$ holds the list of sensors whose child has already been processed. Apart from storing processed sensors, the algorithm utilizes this list to break any cyclic dependency (see dependency among sensors S_6 to S_9 in Figure 5), 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 5 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 5). 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 6 shows the architecture of STDNeut along with the design of its controller. As shown in Figure 6(a), there are two main subsystems of the STDNeut: *(i)* Extended Android Emulator and *(ii)* STDNeut Controller (see Figure 6(b)). We describe the design of the subsystems in this section.

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

STDNeut controller: It is responsible for launching an application 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 [36]. 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 the state of it. 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, AT[1] commands [3] are being utilized. 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, other values are also being fed in response to the AT commands 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. For the hardware/environment events that alter the device state, we use the emulator console to supply realistic data periodically. The Android emulator itself provides most of the hardware like sensors, GPS, signal strength, and others, the data for them can be fed by using emulator console at run-time. 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. In this observation, we have added the BTS information alteration interface through the emulator console, and the STDNeut controller is supplying the realistic BTS identity. We discuss the detailed design of STDNeut controller in the next section.

[1]AT commands are set of instructions that is used to control the modem. Here AT is a short for "attention."

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 $trajectory$

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 6(b), the STDNeut controller 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, then it is impossible that a person can reach New York in 5 min. Hence, a random GPS location alerts an application about the emulated environment. So 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 along with 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 that may vary from 1 to 3 kms depending on the area density in terms of 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 [74]. 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 needed to retrieve BTS ID that operates 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 operation within a country. Now, we only need location area code and cell-ID to get the desired BTS ID, and that is retrieved by calculating the distance with stored BTS location in the database and the current GPS location and compared against the maximum distance allowed. We have used `haversine` [73] to measure the distance between the BTS location and the current GPS location. The main reason for separate module for GPS to BTS correlation is because it requires to interact with external database for retrieving the BTS ID according to the GPS location. In next section, we

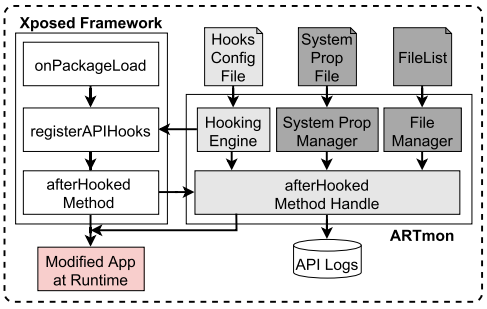


Fig. 7. ARTmon design using Xposed Framework.

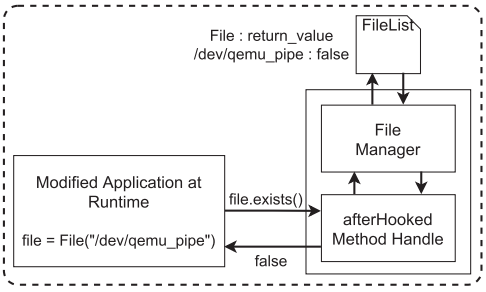


Fig. 8. Working of anti-emulation-detection using ARTmon.

discuss the design of ARTMon that is designed on the top of STDNeut with extended anti-emulation-detection capability that is not supported by the STDNeut.

5 ARTmon: MONITORING FRAMEWORK APIs

ARTmon design is based on the `Droidmon` that is developed using the `Xposed` framework. At a high level, ARTmon can be considered an improved version of `Droidmon`, with more configuration options and built-in anti-emulation-detection techniques related to the files and system properties. Figure 7 illustrates the design of ARTmon using the `Xposed` framework. ARTmon modifies the hooking engine and the hooks config file used by `Droidmon` (Section 2.2.1) to incorporate anti-emulation-detection measures. In ARTmon, hooking engine and associated config file will not only provide the information to instrument framework level APIs but also modifies the return value of an API call as per the configuration. This is useful to provide non-detectable values when system properties and files information is queried from the applications. To provide anti-emulation-detection measures related to system properties and files, ARTmon provides configuration files through which the controller can dictate the values provided to serve application queries. For example, system property manager configuration can contain settings to serve run-time information queries (e.g., whether the platform is a debugger) from the application. Similarly, the file manager can be configured to conceal file related information when queried through APIs like `Exists and Open`.

A user can modify the three configuration files mentioned above to control the application profiling behavior and enable different anti-emulation-detection measures. At runtime, the afterHooked Method Handle, invoked from the modified applications, consults with all managers: system Prop manager, File manager and hooking engine: to substitute the application behavior as specified in the respective configuration files. For some APIs (e.g., open), beforeHooked method is also used.

Figure 8 shows a working example of the anti-emulation-detection techniques supported by ARTmon. As shown in Figure 8, when a modified application (using `Xposed`) requests to check the existence of `/dev/qemu_pipe` file (with the intention of detecting emulation), the afterHooked method handle consults the File manager to check the configuration settings. The file manager checks the file name by looking up the configuration file for the available measures related to anti-emulation-detection through file operations. If the requested file is present in the FilesList config file, then it provides the configured return value to the File manager (in our case: `false`), which is then communicated back to the afterHooked method handle. According to the response received from the file manager, afterHooked method handle substitutes the behavior of the `FileExists` method. Similarly, ARTmon provides defense mechanisms against the emulation-detection attacks through different application methods using user provided configuration files. *Note that ARTmon configuration files Hooks Config File, System Prop Files, and FileList are external files. Users can update these files at any time, which does not require recompilation of the ARTmon module.*

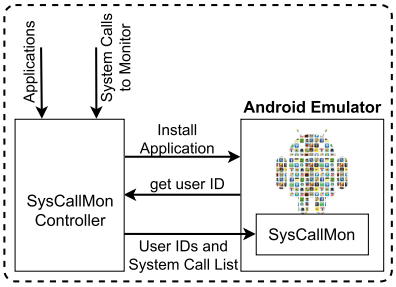


Fig. 9. High-level view of SysCallMon module.

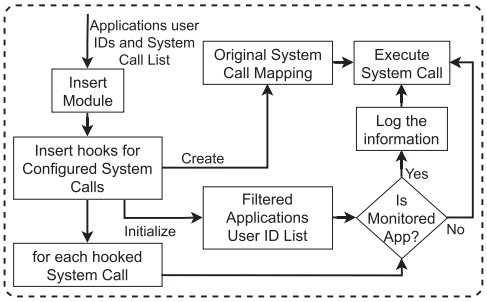


Fig. 10. SysCallMon module work-flow.

6 SysCallMon: SYSTEM CALL MONITOR

SysCallMon is a configurable kernel module designed to monitor and profile the system calls used by applications in an efficient manner. As shown in Figure 9, the SysCallMon controller provides a list of system calls to be profiled for different applications to the SysCallMon through ADB. One of the design challenges is to filter system calls of interest from profiled applications to reduce overheads mentioned in Section 3.2. **Process ID (PID)** comes as the first choice for this purpose, but fails in this case as process ID is allocated on a `fork()` system call from the parent process. By implication, the complete process tree hierarchy is required to be profiled resulting in unnecessary overheads.

In Android systems, applications can be identified easily through user IDs as different applications are associated with different user IDs. Each application is assigned a unique user ID at the time of installation by the Android system. The SysCallMon controller communicates the user ID to the SysCallMon for targeted system call profiling. Next, we discuss implementation aspects including the subtle challenges to design SysCallMon.

6.1 Implementation

As discussed earlier, the SysCallMon is a kernel module designed to profile the system calls invoked by applications in a configurable manner. Figure 10 shows, SysCallMon takes the list of user IDs corresponding to the applications of interest along with the list of system calls to be monitored as input at the time of module initialization. To filter applications under observation, SysCallMon initializes a list containing the user IDs of monitored applications during initialization. Note that, the controller leverages the one-to-one correspondence between applications and the user IDs, and configures the SysCallMon depending on the end-user requirements. Further, during the module initialization, SysCallMon modifies the system call handler table of the Linux kernel to insert modified system call handlers (used as profiling hooks) for configured system calls and stores the original system call handler functions in a mapping table.

Whenever any application invokes a system call, which is monitored (by modifying the system call handler entry), the modified handler in SysCallMon module is invoked. If the user ID of the current context does not match any of the configured user IDs, then the original system call handler is invoked. Otherwise, the system call details including the arguments are logged before executing the original handler. Note that any system call that is not profiled, SysCallMon does not introduce any change to its original behavior in an unmodified system. Further, for monitored system calls originating from applications that are not of interest, very minimal check (UID check) is required before passing them on in the normal execution path.

To use SysCallMon module on an Android emulator platform, there are several challenges as mentioned below.
Loadable kernel module support: Android emulator kernel provided by the Android SDK does not support loadable kernel modules. Therefore, we cannot load our SysCallMon module to profile system calls. To overcome this issue, we have downloaded the Android emulator kernel (Goldfish/Ranchu) source code and compiled it after enabling the loadable kernel module support.

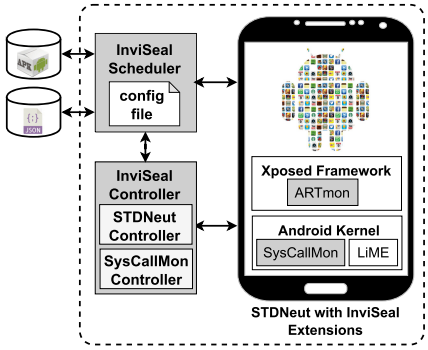


Fig. 11. Architecture of InviSeal, a stealthy dynamic analysis framework for Android systems.

Visibility of system call table: In the recent Linux kernel releases, system call table address is not visible from the user space because of security reasons. Earlier one could read the address of system call table from the `System.map` file, which is located in the boot partition. Android emulator does not have a separate boot partition, and we could not get the address of the system call table. In our custom built kernel, we extract the system call table address by reading the `System.map` file generated during the build process.
System call table page is write protected: In recent Linux kernel, the system call table is mapped as write-protected even for the kernel mode access to avoid accidental overwrite of the original system call handlers. However, write protection does not allow our module to modify system call handlers with our hooks for monitoring system calls. To overcome with this problem, we temporarily disable write protection and reinstate it after modifying the system call handlers for configured system calls. Note that, the same process is repeated to reinstate the original system call handlers during module unload.

7 InviSeal: BUILDING THE SYSTEM

So far, we have provided the anti-emulation-detection technique using the STDNeut (Section 4) and ARTMon (Section 5) to bypass majority of emulation-detection attacks, including sensors, files, and system properties. Further, we have designed an OS-level system call interposition technique to profile the system call activities while incurring low profiling overhead (Section 6). In this section, we provide an overview of the InviSeal, a comprehensive Android security audit framework designed based on STDNeut, ARTMon, and SysCallMon.

7.1 An Overview

InviSeal provides an efficient and easy to use end-to-end Android analysis platform with comprehensive support for anti-emulation-detection. Figure 11 shows, there are three main subsystems of InviSeal: *(i)* STDNeut with InviSeal extensions, *(ii)* InviSeal Controller, and *(iii)* InviSeal Scheduler. We describe the design of the subsystems in this section.
STDNeut with InviSeal extensions is responsible for providing cross-layer profiling capability including the framework layer and the system call layer. ARTmon, designed using the `Xposed` framework to profile the framework layer is one of the important module in the InviSeal. ARTmon along with the STDNeut provides comprehensive anti-emulation-detection functionality to bypass majority of emulation-detection tricks employed by the malware developers. We introduce Linux kernel modifications through a kernel module (referred to as SysCallMon) to profile the system calls in a targeted manner. We also use **Linux Memory Extractor (LiME)** [11] to get the whole memory of an Android device, which can then be analyzed to find useful information like dynamically loaded code or unpacked code of packed malware. LiME is a **loadable kernel module (LKM)**. When we put LiME into the kernel, it writes all of the memory to a file. So, whenever we want to capture the memory of the

system, we need to load the LiME module (insmod) and unload it (rmmod) when the memory capturing process is completed.

InviSeal Controller's responsibilities include launching applications inside the Android virtual device by providing necessary information for application execution and profiling. For example, InviSeal controller provides the hooks configuration file required by ARTMon, system call profiling filters like application user IDs and system call numbers to SysCallMon with the help of SysCallMon controller. To control the profiling characteristics the InviSeal controller makes use of the **Android Debug Bridge (ADB)** [34]. The InviSeal controller uses the Monkey [8] tool to start an application and interact with it. Monkey is a program that runs on every Android device, whether it is an emulator or a real one. It generates pseudo-random user and system events to interact with the application. Apart from launching applications, InviSeal controller also manages the sensors, telephony systems and device state information in Android virtual device with the help of the STDNeut controller. Note that, the ARTMon provides most of the anti-emulation-detection mechanisms except for sensor, telephony system and device state information related emulation-detection that is handled by the STDNeut (see Section 4). This is achieved by frequently feeding-in realistic sensor data from the STDNeut controller to the emulator. To feed the sensor inputs, the STDNeut Controller uses the emulator console [36] APIs. Hence, ARTMon along with STDNeut provides comprehensive anti-emulation-detection functionality to bypass majority of emulation-detection attacks. Other than the core features mentioned above, the InviSeal controller also enables and configures other functionalities like screen-shots, screen recording, network capture, memory dump, and so on.

Memory dump is an on-demand service in InviSeal that must be set up before the analysis can begin. By default, when InviSeal is told to capture system memory, it takes two memory dumps, one at the beginning of the analysis and one at the end. However, InviSeal can also be set up to capture more than two memory dumps, which will be captured simultaneously with equal time intervals while the analysis is going on.

Similar to the memory dump, network capturing is also an on-demand service. To capture network traffic, we use the emulator console, which eventually uses the TCP dump to capture the network traffic.

InviSeal Scheduler provides a single-point dynamic profiling and analysis support to the end-user. The end-users submit different applications for security analysis (referred to as analysis jobs) through external APIs. The main responsibility of the scheduler is to schedule analysis jobs on available computing resources by downloading the applications and provisioning InviSeal controller as per the configuration mentioned in the "config" file. This configuration file "config" is the master "config" file for InviSeal, which is used to configure STDNeut and SysCallMon. The computing resources are managed in the units of **Android Virtual Device (AVD)**, provisioned before an analysis job is scheduled and released when the job finishes. It also captures the profiling information from the InviSeal controller and stores them for further analysis.

8 EVALUATION

To evaluate InviSeal, we created an Android image for an x86-based emulator using the **Android Open Source Project (AOSP-7.1)**. Most Android applications are created for the ARM platform, meaning they cannot operate on other CPU architectures if they contain native code. As a result, the execution of Android applications may fail on an x86-based platform if they contain ARM native code. To overcome the compatibility of applications on x86 emulators, we have utilized the ARM address translation library (Houdini) [2] and integrated it into AOSP-7.1. Intel has designed the Houdini library to execute ARM instruction-based code on an x86 environment. When we load an ARM code on the x86 architecture, the Houdini library executes this ARM code in the x86 environment by translating it into equivalent x86-based instructions. In addition to the AOSP-7.1, we have used a custom-built Android goldfish kernel (v3.10) to integrate and test SysCallMon. 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 SD card along with all the sensors.

Note that we use AOSP-7.1 and goldfish kernel v3.10 for this evaluation. This is needed, because it is closer to the existing analysis frameworks that use Android OS older than 7.1. Furthermore, the choice of 7.1 allows us to

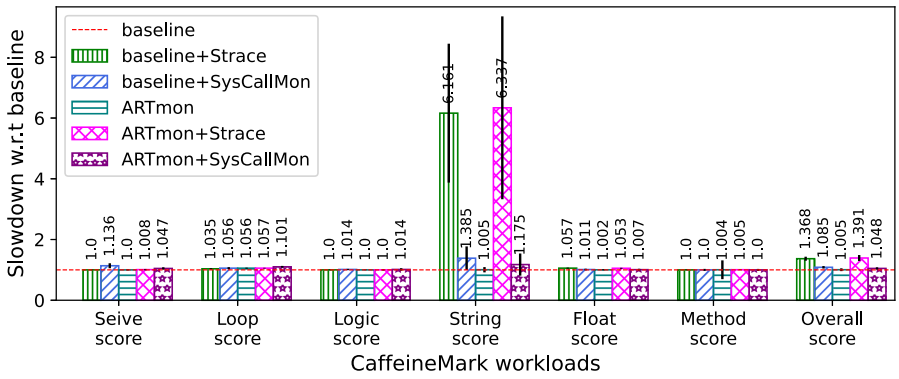


Fig. 12. System slowdown w.r.t baseline (original Android emulator) due to the profiling overheads in different settings using CaffeineMark-3.0 benchmark score. Lower is better.

measure the anti-emulation-detection capabilities of InviSeal and existing analysis frameworks on the same set of real malware and emulation-detection library by making small changes to the samples. The small changes that we made included recompiling malware samples by adjusting lower and higher operatable API versions. A higher version of Android OS and the kernel can also be used with our solution, which requires an appropriate custom-built kernel and AOSP image.

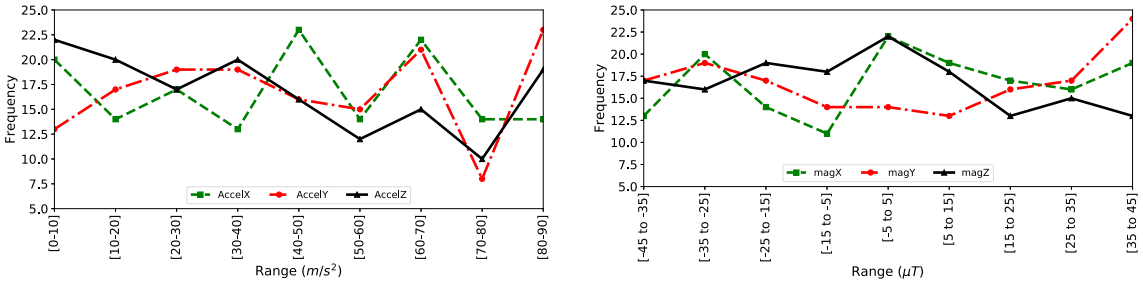
8.1 Performance Overhead Analysis

To empirically analyze performance overhead of InviSeal, we ran CaffeineMark-3.0 [32] with and without ARTmon where `baseline` refers to original emulator without any modifications and `ARTmon` refers to emulator with the proposed framework-level API profiling (Figure 12). Further, we enabled features like `strace`-based profiling and SysCallMon in both the baseline system and with ARTmon to study the additional overheads due to system call monitoring. Note that, `baseline+Strace` refers to a scenario without ARTmon-based profiling but with strace-based system call profiling. The other settings are to be interpreted in a similar manner. We repeated each experiment on a single AVD for ten times and present the average score reported by CaffeineMark, where higher score represents lower overheads.

Figure 12 shows that InviSeal with ARTmon and SysCallMon incurs 1.04× (±0.037 STD) slowdown compared to the baseline system. Only ARTmon without any system call profiling results in similar slowdowns compared to the baseline system. We believe that, `Droidmon` will also result in similar overheads as that of ARTmon as the underlying `Xposed` framework remains the same. When Strace-based system call profiling is enabled, the additional overhead due to strace is 1.36× (±0.066 STD) and 1.38× (±0.098 STD) with baseline and ARTmon, respectively. SysCallMon is ∼1.26× faster than the `strace` in both the cases. This shows that, InviSeal provides cross-layer profiling (using SysCallMon) without incurring any significant additional overheads. Moreover, for benchmark workloads issuing high-volumes of system calls, Strace-based profiling results in significant overheads (e.g., 5.39× overheads in String-score) compared to SysCallMon. *Note that we have not done an overhead study for memory dumps, because we usually take a memory dump before and after an application runs. So, getting the memory dump does not slow down the system. However, let us say we capture the memory while an application runs. In that case, it stops the application from running until the LiME takes over the memory, which incurs performance overhead for the application.*

8.2 Validation of Proposed Anti-Emulation-Detection Measures

InviSeal Versus EmuDetLib/RealMal: We evaluated the effectiveness of the InviSeal against the EmuDetLib (see Section 3.1) and found that InviSeal remains undetected against all the attacks performed by EmuDetLib



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

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

Fig. 13. Effectiveness of InviSeal in neutralizing emulation detection using sensors by providing random reading for accelerometer and magnetometer.

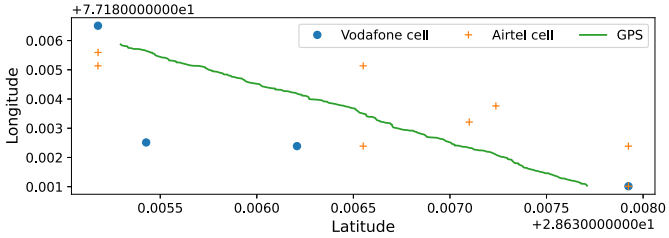


Fig. 14. GPS latitude and longitude reading with anti-emulation measures by feeding-in realistic data along with associated BTS. GPS denotes path trajectory generated using the path patching algorithm.

(see Table 2) to detect the underlying emulated environment. Similarly, when the malware sample RealMal is used, InviSeal also remains undetected against all the real malware samples (see Table 3). After evaluating the efficacy of the InviSeal against the EmuDetLib and RealMal, we attempt to understand the reasoning behind this strong defense mechanism by performing various experiments. In the remaining part of this section, we discuss the reasons for the efficacy of InviSeal by analyzing different sensor readings and device information during the experiments.

8.2.1 Non-detectability through Sensors. To evaluate the efficacy of InviSeal against potential malware exploiting sensor readings, we have developed an application to record and store the values of accelerometer, magnetometer, and GPS readings periodically, which are shown in Figures 13 and 14. 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 13(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 13(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 13(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.

Table 4. **Device Information** Provided by InviSeal and AMS with Three **Different AVDs** Executing the **Same Application**

Queried Information	Information retrieved		
	AVD1	**AVD2**	**AVD3**
PhoneNumber	9876543210	9856543410	9876573213
SimSerialNumber	89914105611117910720	89914145211132510720	89914111219018510720
IMSI	405541385237906	405521385237806	405511385238906
IMEI	359470010002931	359470010302943	359470010002949
SimOperator	40554	40552	40551

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 14, InviSeal anti-emulation-detection measure can provide valid latitude and longitude values along with the associated BTS. In Figure 14, 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.

8.2.2 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, and so on, are either absent or static values are present. To demonstrate the effectiveness of InviSeal's anti-emulation-detection measures, we have used an application called `SIMCardInfo` [42], which extracts the information related to telephony services. We created three instances of this application in three different AVDs and executed all the instances simultaneously for one minute with and without InviSeal. The output of the application queries related to the device information is logged for all instances. We analyzed the log to extract information like IMEI and IMSI. Table 4 shows the captured device information with InviSeal. 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 4, InviSeal 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 sees same device identity. However, the values of PhoneNumber, IMEI, and IMSI are generated manually in the experiment, which can be configured through the InviSeal's configuration file without any modification in the Qemu.

8.3 InviSeal Use Cases

We present three use cases of InviSeal to show its effectiveness as a security analysis platform.

8.3.1 Detection of Collusion Attacks Using Cross Layer Profiling. To show the effectiveness of cross-layer profiling in detection of potential malware, we have developed two applications—*ReadDevInfo* and *SendInfo*. *ReadDevInfo* has permissions to read device information and can write to external storage (in our case the SDCard), whereas *SendInfo* has permissions to access internet and read from external storage (Figure 15). During execution, *ReadDevInfo* queried device information (IMEI, IMSI and Sim Serial Number) using framework level APIs and wrote these information to a file `/sdcard/myfile.txt` using native APIs. *SendInfo* read the same file and established a network connection using socket APIs (part of native library). Note that, *SendInfo* did not use any framework level APIs to read from file or send data across the network.

In this scenario, to detect information leakage through multiple applications, cross-layer profiling is required. Emulator platforms like `Droidmon` provides only framework level activity profiling, which is insufficient to successfully tackle this scenario. In this case, `Droidmon` like profiling techniques can follow the device information trail within the framework layer and get constrained because of their inability to extend the monitoring

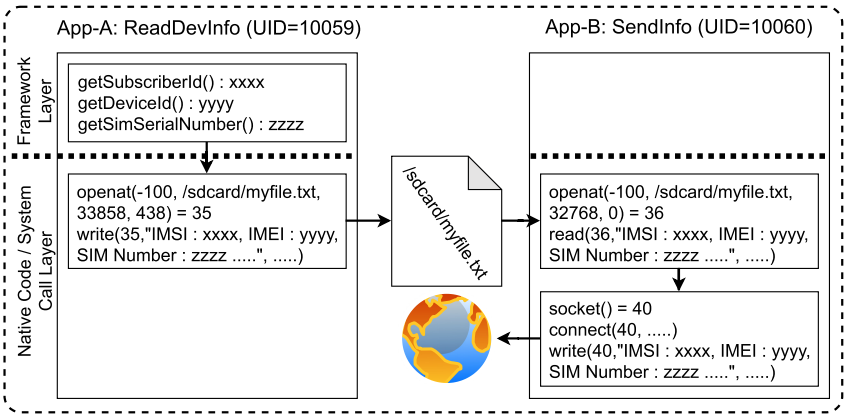


Fig. 15. Detecting collusion attack using cross layer profiling. Dotted lines separate the framework and native layers.

into the lower layers. However, with cross-layer profiling support of InviSeal, we can follow the lead into the captured system calls (by SysCallMon) and flag the malware in a conclusive manner. For example, `openat(''/sdcards/myfile.txt'')` followed by `write()` system call with device information as argument from *ReadDevInfo* is captured by SysCallMon. Additionally, the framework layer also captures the device information API invocation event. When the same file is read from the *SendInfo* application (using `openat` and `read`), the SysCallMon profiles these system calls. A malware analyst becomes suspicious at this point and tracks the other system call made by *SendInfo* to not only detect the malware but also the remote entity involved in this activity by monitoring socket system calls (`socket, connect`). Similar information leakage through shared memory and other native IPC mechanisms can also be easily detected by InviSeal.

8.3.2 Evading Distributed Emulation-Detection. To show the effectiveness of the anti-emulation-detection measures against emulation-detection using multiple clients along with a central server, we used Dendroid [1], a real Android botnet. We integrated EmuDetLib [48] into the Dendroid malware. We modified the Dendroid control server [1] 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. 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 by the control server. We created two instances for each of the CuckooDroid, AMS, and InviSeal, and then executed Dendroid malware with integrated EmuDetLib. 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 InviSeal system instances and not to the instances of CuckooDroid and AMS. The main reason behind this strange behavior was that the unique device information (like phone number, IMEI, and others) provided to the control server by the CuckooDroid and AMS were identical for both the instances of each sandbox, which was not the case with InviSeal. However, when we modified the unqiue device information for each instance of AMS inside the anti-emulation-detection module, the control server started sending instructions about the HTTP flood on the victim side to the instances of AMS. Therefore, we can conclude that the proposed InviSeal system can prevent emulation-detection orchestrated in a distributed setup without any modification, whereas we need to make changes in the anti-emulation-detection module on each instance of AMS to achieve this behavior.

8.3.3 Experiments with Memory Dumps. In this experiment, we have selected a set of 50 random malware detected in year 2018. InviSeal is configured to capture the SysCallMon and ARTmon logs. Also, two memory dumps were captured for each sample. One dump (initial) was taken before the installation of the sample on AVD,

Table 5. Categories Wise Average Difference between Initial and Final Memory Dump

Category	Average number of difference
Process List	7.289
Process XVIEW	3.578
PID hash Table	325.2
NETSTAT	3.11
Libraries	1
Hidden DLLs (ldrmodule)	2340.31
PLT Hooks	0.022
API Hooks	0.022

Table 6. Dynamically Loaded File Extracted from the Memory Dump

App Package Name	File Path	VMA
com.gwjppa.LZstory	/data/app/com.gwjppa.LZstory-1/lib/arm/libSecShell.so	0xc156000
	/data/user/0/com.gwjppa.LZstory/.cache/classes.jar	0x946f1000
com.kuman.comic	/data/user/0/com.kuman.comic/.cache/classes.jar	0x97866000
	/data/app/com.kuman.comic-1/lib/x86/libSecShell.so	x98a62000

while another one (final) was after executing the sample on the device. We have analysed both the memory dump (i.e., initial and final) by obtaining the difference between them with the help of volatility [9] framework and shown in Table 5. During the experiment, 45 samples were able to finish their execution successfully while the remaining 5 could not due to the exceeding time limit, hence the final dump has not been collected. Results shown in Table 5 represents the average difference in each category. We have not shown such categories whose differences are zero, but they can be obtained from the captured memory dumps, if required.

Further, memory dump feature can also be used to extract useful evidences/informations like application loading dynamic binaries into the memory at execution time i.e., dynamic loaded code or unpacked code of a packed malware. To show the ability to extract such information, we have randomly selected two applications where the information about the dynamically loaded code has been captured by the ARTmon. From the ARTmon log, we found that these applications are loading two files, one is the shared library (.so file), and another one is dex/jar file. To extract these files from the memory dump, we have used volatility tool, and extracted from the final memory dump. The path for these files inside the memory dump and the starting virtual memory address (VMA area) is shown in Table 6. An analyst can further perform static analysis to extract more useful information towards the identification of malicious behaviour. Note that the main aim of this experiment is to show that InviSeal can capture the system memory so that memory forensics can be used to find the malicious behavior of non-persistent malware. As it has to do with the whole memory dump, showing the experiment with malware samples from 2018 does not mean that we cannot use the memory forensics feature of InviSeal to look at malware samples from more recent years.

Evaluation summary: In a nutshell, InviSeal can be used to effectively analyse and detect stand-alone and colluding malware in an efficient manner without being detected as an emulated environment.

9 RELATED WORK

Dynamic analysis techniques for Android application analysis have been drawing the interest of researchers for a long time due to the high use of dynamic code loading and other techniques in the application [66]. Because of these techniques, static analysis fails to capture the behavior of an application. Moreover, there are some

dynamic analysis frameworks that have been proposed in the past, which are the main discussion point of this section.

Dynamic analysis tools based on taint analysis such as taintDroid [40] and taintART [64] are capable of identifying the leakage of sensitive information through the framework level API. However, such dynamic analysis techniques are not able to capture the information leak that happen through lower level APIs like native code or system calls.

Apart from the taint analysis, framework level API monitoring-based dynamic analysis tools have also evolved [22, 53, 58, 67]. Such tools can be designed either by modifying the framework level APIs in AOSP or by using hooking frameworks like `Xposed` [37] or FЯIDA [10]. Hooking framework-based approaches are flexible and are easily extendable in comparison to the modification in framework level APIs.

DroidBox [53] is a highly popular and widely used dynamic analysis tool for the Android application and is based on the framework level API modification in AOSP. Droidbox also provides the functionality of taint analysis by including the taintDroid in it. Even though Droidbox is popular among researchers, it can only analyze an application that can run till Android version 4.4. However, the dynamic analysis tools like CuckooDroid [67] and MobSF [58] are based on the `Xposed` (hooking framework) and are capable of hooking the framework level APIs at runtime. These tools use the `Droidmon` [45] module (Section 2.2.1) to enlist the APIs used by an application. MobSF provides a virtual machine for Android x86 version 4.4.2 and can execute only those applications that do not contains native code or requires Android version 5.0 and above. However, CuckooDroid can be used to run on x86-based virtual machines as well as on ARM-based emulated devices. CuckooDroid can be utilized and configured for Android application analysis with Android version 4.4 and above because of the underlying `Xposed` framework. Like the CuckooDroid, AMS (Android Malware Sandbox) [22] is a dynamic analysis framework that uses the FЯIDA framework to monitor/profile framework-level APIs. However, these tools can only capture the information at the framework level and fail to capture transactions happening through the lower level. To overcome this limitation, we can use `strace` utility to capture kernel-level system call information. However, `strace` introduces a very high overhead to capture the system calls made by an application (as discussed in Section 3.2). The primary cause of the huge overhead of `strace` is the underlying `ptrace` system call that `strace` uses to monitor each system call of interest.

Other than the hooking and framework level API modification, **virtual machine introspection (VMI)** [43] techniques have also been employed for dynamic analysis like DroidScope [77], Ndroid [76], and CopperDroid [65]. DroidScope provides functionality to trace native instruction, Dalvik Instruction, and taint tracking. It is built on top of the QEMU emulator and targets Android version 4.3 (Jelly Bean). However, Ndroid is a VMI-based taint analysis tool and can run an application that uses ART as the default runtime. Similarly, CopperDroid [65] tracks the system calls and IPC using the VMI technique and reconstructs the system and high-level Android-specific behavior. CopperDroid can capture lower and framework level behavior effectively using VMI-based techniques. However, the overhead incurred by this approach makes the system more than 10× slower [65, 76, 77], which may reveal the analysis environment to the malware.

Some dynamic analysis frameworks like Afonso et al. [15], Dynalog [18], ANANAS [38], and DroidFAX [30] were also available to profile an application. These frameworks make it possible to track an application on real devices and virtual platforms without making any changes to the Android OS. However, these frameworks change the original application by adding profiling capabilities for framework-level APIs. Given that, these frameworks necessitate modifications to an application, which compromises the integrity of the application. Malware that checks its integrity can bypass the dynamic analysis process. Also, these frameworks do not provide the inherent capability of profiling lower-level information like system calls except for ANANAS and requires the employment of strace utility, which already has issues, as discussed earlier. Like our solution, ANANAS uses the kernel module to record the system calls made by an application, but it is possible that ANANAS could also capture system call information from other applications along with the system call information of the targeted application. This would make the log file bigger and take a long time to parse.

Apart from profiling an application, analysis tools must provide the ability to hide an emulator from the running sample. However, the tools mentioned earlier also use anti-emulation-detection techniques to hide the emulator, but a smart malware author can bypass such a defense mechanism against VM detection. For example, motion sensors can be used to evade dynamic analysis when running on emulated devices [62]. Moreover, none of the analysis frameworks mentioned above provide the ability to bypass the distributed and smart UDI-based emulation-detection attacks. However, to some extent, AMS can bypass these attacks if, for each instance, an updated anti-emulation-detection module is configured manually (as shown in our evaluation).

Besides the open-source analysis frameworks, some commercial sandboxes are also available. One such framework is Joe Sandbox [6], which can perform static and dynamic analysis on the Android platform. It uses a hypervisor-based instrumentation technique similar to VMI and modification in an application to capture different layers of information. It is unknown to us the capability of Joe sandbox to protect the virtualized platform from emulation-detection attacks.

Furthermore, some evasive malware detectors [14, 47] have been proposed that uses both virtual and real environments. These systems look at how the execution traces of an application from different analysis frameworks differ to see if evasion is possible. The main goal of these works is to find evasive malware, whereas we aim to analyze malware only in emulated environments while thwarting the emulation-detection attacks.

10 DISCUSSION AND FUTURE DIRECTIONS

This research presents a new dynamic analysis framework called InviSeal. It uses the emulated platform to profile an application to find malicious intentions while hiding the underneath emulated platform. To validate the efficacy of InviSeal, we use an **emulation-detection library (EmuDetLib)** and real malware samples from the year 2019. From the evaluation, we found that InviSeal remains undetected against the fingerprint-based emulation-detection attacks, while other frameworks cannot hide from one or more than one attack. Now, we discuss more details of InviSeal related to the usage, deployment, impact of the rapid evolution of the Android ecosystem, application and malware, and others, followed by the future directions to extend this work.

Ability to generate realistic data for sensors: InviSeal aims to thwart platform sensing malware that utilizes sensor data to evade dynamic analysis. To make this possible, InviSeal generates realistic sensor values while maintaining their dependencies (Figures 13 and 14). To ensure proper correlation among sensors, InviSeal uses a sensor data generation algorithm (Algorithm 1). However, to generate realistic data, analysts need to define a list of sensors and their dependencies. A sensors-based emulation-detection attack can be avoided in many other ways, such as by using symbolic execution or recording the sensor data from a real device and playing it back on the emulated platform. Symbolic execution requires the interruption in the execution of a program to substitute appropriate value by calculating it on the fly. As a result, symbolic execution slows down the system's performance and opens it up to new forms of evasion attack. The record and replay strategy is still efficient and does not incur extra overhead during analysis. However, it requires recording data for sensors after some time before replaying. Otherwise, an intelligent malware developer may use this information and arm their new malware to thwart the analysis process, which does not happen with our solution. Because in our solution, a user needs to provide two lists, one for available sensors and the other for dependencies among them. Based on these two lists, the sensor data generation algorithm generates the actual data that InviSeal will feed to the emulator at runtime. Therefore, our solution does not need to change the values repeatedly. In case of any alteration in the available sensors list, like adding a new sensor or deleting an existing one, it requires changes in both lists.

Impact of evolution on Android ecosystem and application: The rapid evolution of the Android ecosystem [27] and application [29] design paradigm always imposes new challenges for application developers and security researchers. Moreover, malicious developers also arm malware with advanced techniques [28, 66] such

as dynamic code loading, reflection, native code, or emulation-detection to bypass the analysis processes running on existing systems. This may also be true to some extent with InviSeal. To get around these problems, any analysis system, whether it is static or dynamic, should be able to adapt quickly to such rapid change. As InviSeal is made up of several separate modules, such as STDNeut (which runs Android OS), ARTmon, SysCallMon, and the LiME module, to profile an application across multiple layers. It can easily adapt to the rapid evolution of the Android ecosystem and applications. For example, ARTmon can work on any version of Android OS that is higher than version 7.1, as long as the Xposed framework can be used on that version. In the same way, SysCallMon will also work on another version but requires rebuilding the appropriate Android kernel and the SysCallMon module. However, STDNeut does not need any changes, because it is designed using the Qemu-based Android emulator, which can run any version of Android OS.

Ability to profile collusive malware with cross-layer profiling: One of the use cases of InviSeal is to profile/detect collusive malware that uses either single layer or multiple layers to infiltrate sensitive information in a collusive manner (Section 8.3.1). Many studies [23, 39] effectively target the detection/identification of collusive applications using static or dynamic analysis techniques. The techniques that use only static analysis [39] fail to detect collusive malware that uses both Java framework APIs and native code for collusion attacks. Furthermore, techniques based on model checking [23] by considering the execution state of the application can effectively detect collusive as compared to InviSeal. InviSeal aims to use it in multiple use cases while hiding the underlying emulated platform. It might be that InviSeal is not effective for collusive malware detection as compared to other techniques. However, it can also be used in multiple other scenarios, which is not the case with other collusive malware detection work.

Ability to mitigate new attacks based on files and system properties: The Android ecosystem is ever-evolving. This means that in the future, Android may change or add files and system attributes that might serve as a way of evading analysis. So, as proposed in Reference [26], the system should be adaptable enough to include such modifications without requiring a recompile of the relevant module. In InviSeal, ARTmon provides the defense mechanism against files and system properties-related attacks. Since ARTmon relies on external configuration files, it may be changed at any moment to protect against new threats without rebuilding the ARTmon. However, InviSeal will still be vulnerable to future evasion attempts if such updates in configuration files are not made.

Usage Scenario and Deployment: InviSeal is designed to help analyst perform dynamic analysis of Android applications to find out if they are trying to do anything bad. It is mainly made for an emulated platform to stop platform-sensing malware and profile applications across multiple layers, including framework, native, and kernel. But imagine that analyst want to profile an application on a real device across multiple layers. In that case, it is also possible, given that the device has root access and the Xposed framework can be installed. Also, the analyst needs to recompile the Android kernel on the device so that SysCallMon and the LiME module can be used for system calls profiling and memory dump. There is no need to change the ARTmon to capture framework-level APIs.

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

Availability: We will provide the InviSeal as a GitHub repository that contains an extended Android emulator (STDNeut), SysCallMon kernel module source code, ARTMon, kernel image for Android version 7.1, LiME module binary, along with the controllers. We are preparing the documentation for setting up the InviSeal. We are also making the kernel for the higher version of the Android OS so that an analyst can use them right away without having to rebuild them. We are also making volatility profiles for different Android OS and kernel versions so that memory forensics can be performed easily.

10.1 Future Directions

Similar to the other analysis system, InviSeal is also having some limitations that we like to address in our future works. Some of the limitations and directions for future work are as follows:

(i) Even though InviSeal provides a strong defense against all the malware samples, it falls short if an application tries to detect Xposed framework. For example, the Snapchat application uses the native code to detect Xposed [12]. 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 handle file and system property related emulation-detection attack at kernel-level, which is one of the future directions of this work.

(ii) Malware may leverage the timing channel attacks [26] to bypass the dynamic analysis process on InviSeal. Such timing channel attack includes studying the performance of the graphics sub-system, the number of instructions executed per second, or the mismatch of timing information gathered from an external server and the execution platform. For example, malware can communicate to an external NTP server to get accurate time and measure the time spent on the platform to bypass the analysis process [26]. Therefore, in the future, we like to add the capability of thwarting timing channel attacks into the InviSeal.

(iii) At the moment, InviSeal only dumps information about network traffic as a pcap file and does not have a module to analyze it further. There may be HTTPS/SSL traffic in the captured network information, which makes it harder to get plain text information. So, we would like to add more fine-grained network traffic monitoring utility like MITM-proxy/CharleProxy to store all the information in plain text. We also want to add a network analysis module to gather useful information from the captured traffic.

(iv) Like many other dynamic analysis systems, InviSeal uses the monkey [8] tool to execute a single PATH at a time. Typically, a user's activities will cause an application to change its behavior; the monkey does not have this feature. In the future, we want to enhance InviSeal with the sophisticated input-generating techniques such as Droidbot [55] to circumvent the constraints imposed by the monkey.

11 CONCLUSION

In this article, we have shown that anti-emulation-detection measures provided by the existing dynamic analysis frameworks are insufficient to operate in a stealthy manner resulting in the detection of the underlying platform from malware. Further, we have shown the limitations (both in terms of overhead and stealthiness) of system call monitoring utilities like `strace` to capture the application behavior below the framework layer. To address these issues, we have presented InviSeal, a stealthy dynamic analysis framework for Android systems to perform cross-layer targeted profiling of an application in the virtual environment while hiding the underlying execution environment from the applications. We characterize the effectiveness of the InviSeal by evaluating it with different setups. We have performed experiments to showcase the effectiveness of InviSeal in providing low overhead cross-layer profiling support with comprehensive anti-emulation-detection measures. Further, we have demonstrated the effective use of cross-layer profiling to detect colluding malware. The main aim of InviSeal is to deploy it in the lab environment to carry out Android malware research independently or as a group. To make this possible, we will release InviSeal as a GitHub repository with proper documentation related to deployment and usage.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful and constructive comments.

REFERENCES

[1] GitHub. 2015. *GitHub—qqshow/Dendroid: Dendroid Source Code. Contains Panel and apk.* Retrieved from https://github.com/qqshow/dendroid.

[2] GitHub. 2017. *Rprop/Libhoudini: The Default ARM Translation Layer for x86, Extracted Partly from Nexus Player.* Retrieved from https://github.com/Rprop/libhoudini.

[3] Qt Extended Developer Resources. 2020. *AT Commands—3GPP TS 27.007*. Retrieved from https://doc.qt.io/archives/qtextended4.4/atcommands.html.
[4] IDC Corporate. 2021. *IDC: Smartphone Market Share-OS*. Retrieved from https://www.idc.com/promo/smartphone-market-share/os.
[5] Statista. 2021. *Number of Daily Android App Releases Worldwide|Statista.com*. Retrieved from https://www.statista.com/statistics/276703/android-app-releases-worldwide/.
[6] Joe Sandbox. 2022. *Deep Malware Analysis—Joe Sandbox*. Retrieved from https://www.joesecurity.org/joe-sandbox-technology.
[7] Man7.org. 2022. *Strace(1)—Linux Manual Page*. Retrieved from https://man7.org/linux/man-pages/man1/strace.1.html.
[8] Android Studio. 2022. *UI/Application Exerciser Monkey|Android Developers*. Retrieved from https://developer.android.com/studio/test/other-testing-tools/monkey.
[9] Volatility Foundation. 2022. *The Volatility Foundation—Open-source Memory Forensics*. Retrieved from https://www.volatilityfoundation.org/.
[10] Frida. 2022. *Welcome|Frida • A World-Class Dynamic Instrumentation Framework*. Retrieved from https://frida.re/docs/home/.
[11] 504ensicsLabs. 2021. *LiME Linux Memory Extractor*. Retrieved from https://github.com/504ensicsLabs/LiME.
[12] AeonLucid. 2019. *Snapchat Detection on Android—AeonLucid*. https://aeonlucid.com/Snapchat-detection-on-Android/. Accessed 21 December 2021.
[13] Vitor Afonso, Antonio Bianchi, Yanick Fratantonio, Adam Doupé, Mario Polino, Paulo de Geus, Christopher Kruegel, and Giovanni Vigna. 2016. Going native: Using a large-scale analysis of Android apps to create a practical native-code sandboxing policy. In *Proceedings of the Network and Distributed System Security Symposium*. 1–15.
[14] Vitor Afonso, Anatoli Kalysch, Tilo Müller, Daniela Oliveira, André Grégio, and Paulo Lício de Geus. 2018. Lumus: Dynamically uncovering evasive android applications. In *Information Security*, Liqun Chen, Mark Manulis, and Steve Schneider (Eds.). Springer International Publishing, Cham, 47–66.
[15] Vitor Monte Afonso, Matheus Favero de Amorim, André Ricardo Abed Grégio, Glauco Barroso Junquera, and Paulo Lício de Geus. 2015. Identifying Android malware using dynamically obtained features. *J. Comput. Virol. Hack. Techniq.* 11, 1 (2015), 9–17.
[16] G DATA Software AG. 2019. *A New Malware Every 7 Seconds|G DATA*. Retrieved from https://www.gdatasoftware.com/news/2018/07/30950-a-new-malware-every-7-seconds.
[17] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting millions of Android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*. 468–471.
[18] Mohammed K. Alzaylaee, Suleiman Y. Yerima, and Sakir Sezer. 2016. Dynalog: An automated dynamic analysis framework for characterizing android applications. In *Proceedings of the International Conference On Cyber Security And Protection Of Digital Services (Cyber-Security'16)*. 1–8. https://doi.org/10.1109/CyberSecPODS.2016.7502337
[19] Android Developers. 2021. *Run Apps on the Android Emulator|Android Developers*. Retrieved from https://developer.android.com/studio/run/emulator.
[20] android.com. 2019. *Android*. Retrieved from https://www.android.com/.
[21] Sebastian Bachmann, Anthony Desnos, and Geoffroy Gueguen. 2021. *Welcome to Androguard's Documentation!—Androguard 3.4.0 Documentation*. Retrieved from https://androguard.readthedocs.io/en/latest/.
[22] Areizen. 2020. *Android Malware Sandbox*. Retrieved from https://github.com/Areizen/Android-Malware-Sandbox.
[23] Irina Mariuca Asavoae, Jorge Blasco, Thomas M. Chen, Harsha Kumara Kalutarage, Igor Muttik, Hoang Nga Nguyen, Markus Roggenbach, and Siraj Ahmed Shaikh. 2016. Towards Automated Android App Collusion Detection. Retrieved from https://arxiv.org/abs/1603.02308. https://doi.org/10.48550/ARXIV.1603.02308
[24] Michael Backes, Sven Bugiel, Oliver Schranz, Philipp Von Styp-Rekowsky, and Sebastian Weisgerber. 2017. ARTist: The android runtime instrumentation and security toolkit. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroSP'17)*. 481–495. https://doi.org/10.1109/EuroSP.2017.43
[25] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference (ATEC'05)*. 41.
[26] Marcus Botacin, Paulo Lício De Geus, and André Grégio. 2018. Who watches the watchmen: A security-focused review on current state-of-the-art techniques, tools, and methods for systems and binary analysis on modern platforms. *ACM Comput. Surv.* 51, 4, Article 69 (July 2018), 34 pages. https://doi.org/10.1145/3199673
[27] Haipeng Cai. 2020. Embracing mobile app evolution via continuous ecosystem mining and characterization. In *Proceedings of the 7th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft'20)*. Association for Computing Machinery, 31–35. https://doi.org/10.1145/3387905.3388612
[28] Haipeng Cai, Xiaoqin Fu, and Abdelwahab Hamou-Lhadj. 2020. A study of run-time behavioral evolution of benign versus malicious apps in Android. *Info. Softw. Technol.* 122 (2020), 106291. https://doi.org/10.1016/j.infsof.2020.106291
[29] Haipeng Cai and Barbara Ryder. 2021. A longitudinal study of application structure and behaviors in android. *IEEE Trans. Softw. Eng.* 47, 12 (2021), 2934–2955. https://doi.org/10.1109/TSE.2020.2975176
[30] Haipeng Cai and Barbara G. Ryder. 2017. DroidFax: A toolkit for systematic characterization of android applications. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'17)*. 643–647. https://doi.org/10.1109/ICSME.2017.35

[31] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, and Giovanni Vigna. 2017. Obfuscation-resilient privacy leak detection for mobile apps through differential analysis. In *Proceedings of the 24th Symposium on Network and Distributed System Security (NDSS'17)*.

[32] Pendragon Software Corporation. 2006. *CaffeineMark 3.0 Benchmark Information*. Retrieved from http://www.benchmarkhq.ru/cm30/info.html.

[33] Valerio Costamagna and Cong Zheng. 2016. ARTDroid: A virtual-method hooking framework on android ART runtime. In *Proceedings of the International Workshop on Innovations in Mobile Privacy and Security co-located with the International Symposium on Engineering Secure Software and Systems (IMPS@ESSoS'16) (CEUR Workshop Proceedings, Vol. 1575)*. CEUR-WS.org, 20–28.

[34] Android Developers. 2021. *Android Debug Bridge (adb)|Android Developers*. Retrieved from https://developer.android.com/studio/command-line/adb.

[35] Android Developers. 2021. *Platform Architecture|Android Developers*. Retrieved from https://developer.android.com/guide/platform/.

[36] Android Developers. 2021. *Send Emulator Console Commands|Android Developers*. https://developer.android.com/studio/run/emulator-console.

[37] XDA Developers. 2018. *Xposed Framework Hub*. Retrieved from https://www.xda-developers.com/xposed-framework-hub/.

[38] Thomas Eder, Michael Rodler, Dieter Vymazal, and Markus Zeilinger. 2013. ANANAS—A framework for analyzing android applications. In *Proceedings of the International Conference on Availability, Reliability and Security*. 711–719. https://doi.org/10.1109/ARES.2013.93

[39] Karim O. Elish, Haipeng Cai, Daniel Barton, Danfeng Yao, and Barbara G. Ryder. 2020. Identifying mobile inter-app communication risks. *IEEE Trans. Mobile Comput.* 19, 1 (2020), 90–102. https://doi.org/10.1109/TMC.2018.2889495

[40] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2014. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Trans. Comput. Syst.* 32, 2, Article 5 (June 2014), 29 pages. https://doi.org/10.1145/2619091

[41] Anam Fatima, Saurabh Kumar, and Malay Kishore Dutta. 2021. Host-server-based malware detection system for android platforms using machine learning. In *Advances in Computational Intelligence and Communication Technology*, Xiao-Zhi Gao, Shailesh Tiwari, Munesh C. Trivedi, and Krishn K. Mishra (Eds.). Springer, Singapore, 195–205.

[42] Harry Gonzalez. 2021. *SIM Card Info—Apps on Google Play*. Retrieved from https://play.google.com/store/apps/details?id=me.harrygonzalez.simcardinfo&hl=en_IN.

[43] Y. Hebbal, S. Laniepce, and J. Menaud. 2015. Virtual machine introspection: Techniques and applications. In *Proceedings of the 10th International Conference on Availability, Reliability and Security*. 676–685. https://doi.org/10.1109/ARES.2015.43

[44] Martin Henze, Jan Pennekamp, David Hellmanns, Erik Mühmer, Jan Henrik Ziegeldorf, Arthur Drichel, and Klaus Wehrle. 2017. CloudAnalyzer: Uncovering the cloud usage of mobile apps. In *Proceedings of the 14th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous'17)*. Association for Computing Machinery, 262–271. https://doi.org/10.1145/3144457.3144471

[45] idanr. 2014. *Droidmon: Dalvik Monitoring Framework for CuckooDroid*. Retrieved from https://github.com/idanr1986/droidmon.

[46] Yiming Jing, Ziming Zhao, Gail-Joon Ahn, and Hongxin Hu. 2014. Morpheus: Automatically generating heuristics to detect android emulators. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC'14)*. Association for Computing Machinery, 216–225. https://doi.org/10.1145/2664243.2664250

[47] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. 2014. BareCloud: Bare-metal analysis-based evasive malware detection. In *Proceedings of the 23rd USENIX Security Symposium (USENIXSecurity'14)*. USENIX Association, 287–301. Retrieved from https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/kirat.

[48] Saurabh Kumar, Debadatta Mishra, Biswabandan Panda, and Sandeep K. Shukla. 2020. STDNeut: Neutralizing sensor, telephony system and device state information on emulated android environments. In *Cryptology and Network Security*, Stephan Krenn, Haya Shulman, and Serge Vaudenay (Eds.). Springer International Publishing, Cham, 85–106.

[49] Saurabh Kumar, Debadatta Mishra, Biswabandan Panda, and Sandeep Kumar Shukla. 2021. DeepDetect: A practical on-device android malware detector. In *Proceedings of the IEEE 21st International Conference on Software Quality, Reliability and Security (QRS'21)*. 40–51. https://doi.org/10.1109/QRS54544.2021.00015

[50] Saurabh Kumar, Debadatta Mishra, and Sandeep Kumar Shukla. 2021. Android malware family classification: What works – API calls, permissions or API packages? In *Proceedings of the 14th International Conference on Security of Information and Networks (SIN'21)*, Vol. 1. 1–8. https://doi.org/10.1109/SIN54109.2021.9699322

[51] Saurabh Kumar and Sandeep Kumar Shukla. 2020. *The State of Android Security*. Springer, Singapore, 17–22. https://doi.org/10.1007/978-981-15-1675-7_2

[52] Argus Lab. 2021. *Argus SAF —Argus-PAG*. Retrieved from http://pag.arguslab.org/argus-saf.

[53] Patrik Lantz. 2011. *An Android Application Sandbox for Dynamic Analysis*. Master's thesis. Retrieved from https://www.eit.lth.se/sprapport.php?uid=595.

[54] Li Li, Daoyuan Li, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. 2017. Understanding android app piggybacking: A systematic study of malicious code grafting. *IEEE Trans. Info. Forens. Secur.* 12, 6 (2017), 1269–1284. https://doi.org/10.1109/TIFS.2017.2656460

[55] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: A lightweight UI-guided test input generator for android. In *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C'17)*. IEEE Press, 23–26. https://doi.org/10.1109/ICSE-C.2017.8

[56] Michael Hale Ligh, Andrew Case, Jamie Levy, and Aaron Walters. 2014. *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory* (1st ed.). Wiley Publishing.

[57] Satoshi Maruyama, Katsuhiko Tanahashi, and Takehiko Higuchi. 2002. Base transceiver station for W-CDMA system. *Fujitsu Sci. Tech. J.* 38 (1 2002), 167–173.

[58] MobSF Team. 2020. *1. Documentation Âů MobSF/Mobile-Security-Framework-MobSF Wiki*. Retrieved from https://github.com/MobSF/Mobile-Security-Framework-MobSF/wiki/1.-Documentation.

[59] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. 2009. A fistful of red-pills: How to automatically generate procedures to detect CPU emulators. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies (WOOT'09)*. USENIX Association, 2.

[60] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2014. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *Proceedings of the 21st Network and Distributed Systems Security Symposium (NDSS'14)*.

[61] Alireza Sadeghi, Hamid Bagheri, Joshua Garcia, and Sam Malek. 2017. A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software. *IEEE Trans. Softw. Eng.* 43, 6 (2017), 492–530. https://doi.org/10.1109/TSE.2016.2615307

[62] securityfair.com. 2019. *Android Apps use the Motion Sensor to Evade Detection and Deliver Anubis MalwareSecurity Affairs*. Retrieved from https://securityaffairs.co/wordpress/80037/malware/android-apps-motion-sensor.html.

[63] Mingshen Sun, Xiaolei Li, John C. S. Lui, Richard T. B. Ma, and Zhenkai Liang. 2017. Monet: A user-oriented behavior-based malware variants detection system for android. *Trans. Info. For. Sec.* 12, 5 (May 2017), 1103–1112. https://doi.org/10.1109/TIFS.2016.2646641

[64] Mingshen Sun, Tao Wei, and John C. S. Lui. 2016. TaintART: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*. Association for Computing Machinery, 331–342. https://doi.org/10.1145/2976749.2978343

[65] Kimberly Tam, Aristide Fattori, Salahuddin Khan, and Lorenzo Cavallaro. 2015. Copperdroid: Automatic reconstruction of android malware behaviors. In *Proceedings of the 22nd Network and Distributed Systems Security Symposium (NDSS'15)*. 1–15.

[66] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. 2017. The evolution of android malware and android analysis techniques. *ACM Comput. Surv.* 49, 4, Article 76 (Jan. 2017), 41 pages. https://doi.org/10.1145/3017427

[67] Checkpoint Software Technologies. 2021. *CuckooDroid Book*. Retrieved from https://cuckoo-droid.readthedocs.io/en/latest/.

[68] thehackernews.com. 2019. *New Android Malware Apps Use Motion Sensor to Evade Detection*. Retrieved from https://thehackernews.com/2019/01/android-malware-play-store.html.

[69] Teller Tomer and Hayon Adi. 2014. Enhancing automated malware analysis machines with memory analysis. In *Blackhat Arsenal*. 1–5.

[70] Timothy Vidas and Nicolas Christin. 2014. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIA CCS'14)*. Association for Computing Machinery, 447–458. https://doi.org/10.1145/2590296.2590325

[71] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2018. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Trans. Priv. Secur.* 21, 3, Article 14 (Apr. 2018), 32 pages. https://doi.org/10.1145/3183575

[72] Wikipedia. 2020. *GSM Cell ID*. Retrieved from https://en.wikipedia.org/wiki/GSM_Cell_ID.

[73] Wikipedia. 2020. *Haversine Formula*. Retrieved from https://en.wikipedia.org/wiki/Haversine_formula.

[74] Wikipedia. 2020. *OpenCellID*. Retrieved from https://en.wikipedia.org/wiki/OpenCellID.

[75] Carsten Willems, Ralf Hund, Andreas Fobian, Dennis Felsch, Thorsten Holz, and Amit Vasudevan. 2012. Down to the bare metal: Using processor features for binary analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC'12)*. Association for Computing Machinery, 189–198. https://doi.org/10.1145/2420950.2420980

[76] Lei Xue, Chenxiong Qian, Hao Zhou, Xiapu Luo, Yajin Zhou, Yuru Shao, and Alvin T. S. Chan. 2019. NDroid: Toward tracking information flows across multiple android contexts. *IEEE Trans. Info. Forensics Secur.* 14, 3 (Mar. 2019), 814–828. https://doi.org/10.1109/TIFS.2018.2866347

[77] Lok Kwong Yan and Heng Yin. 2012. DroidScope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX Conference on Security Symposium (Security'12)*. USENIX Association, 29.

Received 11 February 2022; revised 8 September 2022; accepted 22 September 2022