Unmasking the Veiled: A Comprehensive Analysis of Android Evasive Malware

Antonio Ruggia, Dario Nisi, Savino Dambra, Alessio Merlo, Davide Balzarotti, Simone Aonzo

Unmasking the Veiled: A Comprehensive Analysis of Android Evasive Malware

Antonio Ruggia*, Dario Nisi†, Savino Dambra†, Alessio Merlo‡, Davide Balzarotti†, Simone Aonzo†
* University of Genoa, Italy
† EURECOM, Sophia Antipolis, France
‡ Centre for Higher Defence Studies (CASD), Rome, Italy

ABSTRACT

Since Android is the most widespread operating system, malware targeting it poses a severe threat to the security and privacy of millions of users and is increasing from year to year. The response from the community was swift, and many researchers have ventured to defend this system. In this cat-and-mouse game, attackers pay special attention to flying under the radar of analysis tools, and the techniques to understand whether their app is under analysis have become more and more sophisticated. Moreover, these *evasive techniques* are also adopted by benign apps to deter reverse engineering, making this phenomenon pervasive in the Android app ecosystem.

While the scientific literature has proposed many evasive techniques and investigated their impact, one aspect still needs to be studied: how and to what extent Android apps, both malware and goodware, use such controls. This paper fills this gap by introducing a comprehensive taxonomy of evasive controls for the Android ecosystem and a proof-of-concept app that implements them all. We release the app as open source to help researchers and practitioners to assess whether their app analysis systems are sufficiently resilient to known evasion techniques. We also propose *DroidDungeon*, a novel probe-based sandbox, which circumvents evasive techniques thanks to a substantial engineering effort, making the apps under analysis believe they are running on an actual device. To the best of our knowledge, currently, *DroidDungeon* is the only solution providing anti-evasion capabilities, maintainability, and scalability at once.

Using our sandbox, we studied evasive controls in both benign and malicious Android apps, revealing insights about their purpose, differences, and relationships between evasive controls and packers/protectors. Finally, we analyzed how the execution of an app differs depending on the presence or absence of evasive countermeasures. Our main finding is that 14% and 4% of malicious and benign samples refrain from running in an analysis environment that does not correctly mitigate evasive controls.

CCS CONCEPTS

• **Security and privacy** → *Software and application security*; *Malware and its mitigation.*

1 INTRODUCTION

Android is the world's most popular mobile operating system, with over 2 billion active devices, making it a prime target for malware authors [6]. Over the past years, Android malware has significantly evolved in terms of its capabilities, sophistication, and adoption of evasive techniques [58, 79].

In this paper, we study the evasive behavior of Android apps with a focus on techniques used for detecting different forms of dynamic analysis. While these techniques are prevalent among malware to avoid exposing the malicious behavior inside an analysis environment, they are also adopted by benign apps to protect their code from reverse engineering and specific client-side attacks or to ensure that the users' sensitive data (e.g., bank access tokens) are not stored in a rooted device [29, 66, 83]. Either way, the goal of evasive controls is to protect the apps (whether benign or malicious) by retrieving precise and accurate information on the hardware and software components of the system they are running on.

Dealing with this topic is as vast as it is complex. First, we aim to study known evasion techniques without attempting to detect new ones or look for those still unknown. As **first contribution**, we collected all documented evasive techniques by searching through blog posts, malware writeups, and scientific papers, and we categorized them into two main groups. *Direct* evasive techniques (DETs) retrieve specific data that can be directly used to detect whether the app is executed inside an analysis environment. Conversely, *indirect* evasive techniques (IETs) return data that must be further processed to be used for evasion. This distinction is crucial for detecting evasion attempts: a sample employing DETs is unequivocally looking for information on the runtime environment, which per se is enough to infer an evasion attempt, whereas merely employing IETs may serve the same objective, but not necessarily.

Our **second contribution** is the development of a proof-of-concept Android app implementing all the collected evasive techniques we gathered. We were inspired by Al-Khaser [9], an executable for the Windows OS developed to test the stealthiness of sandboxes, which has also been used in several scientific papers to study the Windows evasive malware [37, 44, 63] phenomenon.

Then, to measure the techniques used in the wild, we needed a sandbox to execute both benign and malicious samples. In the context of malware analysis, the term sandbox generally refers to a dynamic analysis tool that runs in a safe and controlled environment for analyzing and observing suspicious code behavior without risking damage to the host or the network [55]. Sandboxes may offer several advantages, such as easily restoring the environment after analysis, ensuring that any malicious activity stays confined to the sandbox, scalability, and replication for a wide range of configurations, making testing the behavior of several suspect samples across different scenarios easier. In [30], the authors highlighted the requirements that a malware analysis sandbox for Android should follow. We can summarize them in two criteria: *anti-evasion* (or "resilience to the detection") and *maintainability*. The former refers to the ability of a sandbox to be transparent to the apps running inside it, creating the minimum possible set of artifacts that allow an app to detect the sandbox itself. Ideally, a sandbox also exposes realistic and consistent information; otherwise, malware may leverage non-coherent knowledge to build a novel evasive technique. Maintainability quantifies developers' effort to address new evasive controls and upgrade the sandbox with a new version of the Android (kernel and OS). Achieving maintainability does not necessarily require avoiding kernel and OS modifications as long as they are easily portable to the newer versions. Another critical aspect of large-scale malware analysis is *scalability*, which measures the system's ability to analyze many samples simultaneously in an automated fashion and under several device configurations.

In Android, sandboxes are often implemented as emulators [96] or as containers [86]. However, at the time of writing, *none of the current Android sandboxes meet the anti-evasion, maintainability, and scalability criteria at once.* Some fully emulated sandboxes (e.g., DroidScope and CopperDroid) cannot offer stealthiness or transparency, leaving several artifacts allowing an app to identify their presence easily. On top of that, these solutions are also based on obsolete versions of Android. Container-based sandboxes, such as VPBox, on the other hand, claim to be resilient to evasion by running on actual phones (bare metal). This, however, comes at the expense of scalability because smartphones' hardware still needs to be improved in computing power, allowing only a limited number of analysis containers to run in parallel on the same phone, and making container-based sandboxes unfit for large-scale measurements or part of heavy-load analysis pipelines. Finally, all the state-of-the-art solutions are not easily upgradable with the latest Android OS versions, significantly reducing their maintainability.

To fill this gap, as a **third contribution** of this paper, we present the design of a new sandbox – named *DroidDungeon*– that jointly meets the anti-evasion, scalability, and maintainability requirements and enables us to perform the first analysis of evasive controls in malware and goodware. Our sandbox's design allows the deployment in both an actual device and an emulated environment, ensuring, in any case, a high level of transparency.

In short, *DroidDungeon* relies on kernel and user probes to monitor the apps' behavior and return "fake responses" from system calls and framework APIs to hide the underlying emulator and bypass the evasion checks. While simple on paper, providing fake responses hides several technical challenges, including parsing and modifying Java objects from outside the managed runtime. Moreover, to avoid side effects, *DroidDungeon* has to modify only the events triggered by the app's logic and not those that ensure the correct functioning of Android. Therefore, it understands *when* to enforce the anti-evasion by performing a custom stack unwind to determine a system or API *call provenance* and precisely determine whether the app under analysis generated a particular event.

Finally, the **fourth (and main) contribution** of our work is using *DroidDungeon* to perform the first study of the actual usage of evasive checks in benign and malicious Android apps. We carefully selected *20,556* malicious and *21,154* apps for our experiments. Malware samples are uniformly spread over 200 different families collected by the VirusTotal [92] live feed until April 2023. Benign applications were retrieved directly from the Google Play store, with a limit of 500 apps for each available category. We release this dataset to the community reporting the samples' hashes (for legal reasons, we do not share the actual samples).

Our measurement aims to answer the following three main questions. The reader will find the answers in Section 6.

RQ1: How do malware and goodware differ in using evasive techniques?

RQ2: What is the relation between evasive controls and packers/protectors?

RQ3: Which operations are hidden under evasive controls?

What we discovered is extremely interesting. Malware mainly leverages evasive controls to verify the environment in which they are executed; in particular, almost 70% of evasive malware aims to detect the emulated environment. We also detected one malicious sample that leverages the SafetyNet [51] Attestation API to verify the legitimacy of the environment. This could be a critical tipping point because comprehensive remediation for SafetyNet (and the new Play Integrity API) does not exist. It also shows how "benign" services can be abused maliciously. Our experiments also show that the evasiveness of malware heavily depends on its family: different malware families implement different numbers and types of evasive techniques.

Benign samples are instead more prone to check app-specific features to protect themself from client-side attacks. For instance, more than 88% of evasive goodware verifies from where they were installed, and 82% implements at least one IET control related to signature verification. Moreover, while malware uses evasion techniques predominantly at the beginning, goodware often spreads them over its entire execution.

Moreover, we checked if a relationship between evasive controls and packers exists. We discovered that the presence of a packer does not affect the number of evasive checks but rather the sample's techniques. In particular, packer samples are more prone to check process artifacts than non-evasive ones.

Finally, we analyzed every sample two times: in the first run, *DroidDungeon* hides the underlying emulator by enforcing the anti-evasion criterion, while in the second, the app is executed and monitored without modifying the emulator's behavior. The results highlight that evasive samples perform different events depending on when they are executed. In particular, 14% of evasive malware samples stop their execution before launching any activity when executed in an emulated environment. Also, evasive malware is more prone to interact with potentially dangerous APIs (e.g., record audio and video of the device) or execute CLI commands if executed in an environment that behaves like an actual device.

2 BACKGROUND

This section provides the necessary technical background for the rest of the paper. We start by presenting the relevant details of how an Android app is built, the Android RunTime system, and then introduce the Zygote process.

Anatomy of an Android app. Each Android app is distributed and installed as an Android PacKage (APK) file. In a nutshell, an APK file is a ZIP archive containing all the necessary files to run the first execution of the app, i.e., compiled code, resources (e.g., images for the user interface), and a *Manifest* file. An Android app is usually developed in Java or Kotlin and compiled into the Dalvik bytecode (DEX file). In addition, Android apps could also include C/C++ code, which is compiled into a native library (SO file) for each supported architecture. The Manifest provides valuable information about the app, particularly its package name (on Android, you cannot install two apps with the same name), its components, and the required permissions.

At installation time, Android creates a dedicated directory for each app in which the system copies the original APK, renaming it as *base.apk*. To ensure its integrity, each APK is signed with the developer's private key and contains the corresponding public certificate of the developer. Thus, during the installation process, the Android OS verifies the integrity of the APK and its resources. It is worth noticing that this mechanism does not provide any authentication, as the developer certificate does not need to be issued by any trusted certificate authority.

Android ART. Since Android 5.0, Google introduced the Android RunTime (ART) to replace the Dalvik virtual machine. While Dalvik just-in-time (JIT) compiles framework code and apps on demand, ART uses a hybrid approach that combines Ahead-Of-Time (AOT), JIT compilation, and profile-guided compilation. When an app is installed, AOT compilation is performed for only a subset of the app's methods. After the first execution and when a device charges, ART performs the AOT compilation of all frequently used code based on a profile generated during the first runs. Thus, during the next executions, ART uses the profile-guided code and avoids doing JIT compilation at runtime for methods already compiled. Methods that get JIT-compiled during the new runs are added to the profile, which the following compilation will pick up. It is worth noting that ART performs AOT compilation also of the framework libraries. However, in this case, the amount of compiled code depends on configuration options specified when the framework is built [48].

The AOT compiled code is saved in a special file format named OAT, a custom ELF executable that includes two special sections: `oatdata` to store headers and info about the compiled DEX files, and `oatexec` to store the compiled code. It is worth noticing that the OAT file format changes between Android versions, and no official documentation tracks those changes. Using AOT compilation has the advantage of achieving better performance at the price of having longer installation times and more extensive storage requirements. As these disadvantages are minor on today's hardware, it is clear why ART was preferred to the Dalvik JIT.

Zygote & app startup. Zygote is the parent process of all Android apps, created by the init process during the system boot. This process initializes the first instance of Dalvik Virtual Machine (DVM) and pre-loads all framework classes that the apps should use very often. Each new app process is a fork of the Zygote process, which is used as a template, thus, saving the time required to load these resources into its address space. In this way, the memory addresses the space of the Android framework, and the native libraries are the same for all Android apps (inherited from the Zygote).

3 TAXONOMY OF ANDROID EVASIVE CONTROLS

An evasive control is a technique that is used by an app to prevent the runtime inspection and analysis of its behavior. This is what the MITRE Malware Behavior Catalog [20] classifies as a malware objective under the name of *Anti-Behavioral Analysis*, and it is important to distinguish it from other forms of code protection (e.g., code obfuscation, encryption, and Dynamic Code Loading) that are instead classified as *Anti-Static Analysis*[1] While the anti-behavioral techniques aim at concealing the action performed by a sample, anti-static analysis techniques focus on protecting the app code by making it more complex to analyze. As such, they do not affect the app's runtime behavior, which continues to execute similarly in any environment. Therefore, the two objectives, evasive controls, and static analysis protection, are orthogonal and often combined together [75]. For instance, Denuvo Mobile Game Protection jointly uses evasive checks (e.g., anti-debugging) and DCL to protect Android apps from repackaging [54].

Over the years, researchers studied different aspects related to evasive controls, such as their adoption in benign samples [29, 33, 66, 87] or their impact on malware classification [22, 27, 28, 32, 38, 64, 65, 68, 72, 89]. Moreover, novel and more sophisticated anti-behavioral techniques have been routinely presented year after year [1–4, 7, 16, 39, 56, 58, 79, 80, 83–85, 91, 98].

A first attempt to propose a taxonomy of protection techniques used in Android apps was recently published in 2023 by Faruki et al. [41]. However, the authors considered only a small subset of the techniques outlined in this paper, focusing mainly on obfuscation. As we already explained, we follow the MITRE classification instead and therefore do not consider obfuscation as part of evasive checks.

By following the MITRE jargon, there are many ways to achieve the same objective; each called a malware *'Behavior'*. Each behavior can be implemented in multiple ways, which MITRE call *'Methods'*.

In the rest of this section, we present the list of evasive behaviors covered in our study by grouping them into three macro-categories: *Environment verification*, *APK tampering verification*, and *High-level verification*.

3.1 Environment verification

Environment checks aim to detect the reliability of the environment in which apps are installed.

Root detection. The Android design does not require users to use the root account; therefore, such an account is disabled by default. A method known as *rooting* allows an end user to get super-user access to an Android smartphone. Super-users can alter system settings, access private areas in the primary memory, install specialized apps, or use a debugger or dynamic analysis tool. Therefore, the presence of executables that require root permissions may indicate that a sample is executed in an instrumented environment, such as a sandbox. Of all the possible tests that can be used for root detection, the most common look for the presence in the file system of the `su` or `busybox` executables or test whether well-known paths that are usually read-only have write permission [81, 83, 87].

Debugging detection. A debugger introduces changes to the memory space of the target process and may impact the execution time of certain code snippets [29, 66]. Thus, anti-debugging techniques can detect (by looking for specific artifacts or side effects) or prevent the app from being debugged.

Hook detection. Anti-hooking controls aim to detect dynamic binary instrumentation tools (e.g., Xposed [52] and Frida [70]) that

[1]The Malware Behavioral Catalog currently focuses mainly on Windows malware and does not contain specific checks for Android. However, we believe it is useful to adopt its naming scheme to present our work better.

can hook and tamper with the execution flow of an app. The simplest way to detect their presence is by scanning package names, files, or binaries and looking for well-known frameworks' resources. In addition, each dynamic analysis framework works differently and may require specific detection techniques. For instance, Xposed is an Android app that applies modules directly to the Android OS ROM and requires root privileges. Contrary, Frida injects instead a JavaScript engine into the instrumented process.

Emulator detection. Anti-emulation techniques try to detect whether the app is running on an actual device. For instance, the Android emulator is built on top of the QEMU [31] emulator and an emulator may not provide the same hardware functionalities as a real phone (for instance, for sensors like gyroscope and accelerometer), or some particular artifacts may or may not be present (e.g., different files or file content, Android system properties). In addition, some emulators do not fully support the Google Play Services (e.g., Genymotion [46]) or require some changes in the system property (e.g., `ro.build.tag`).

Memory integrity verification. This type of evasive control aims to verify the integrity of the app's memory space against memory patches applied at runtime [83]. For instance, hooks to C/C++ code can be installed by overwriting function pointers in memory or patching parts of the function code (e.g., inline hooks that modify the function prologue). Thus, an app can check the integrity of its memory regions to detect any alteration.

App-level virtualization detection. Android virtualization is a recent technique that enables an app (*container*) to create a virtual environment in which other apps (*plugins*) can run while fully preserving their functionalities [61, 90]. The container app acts like a proxy, intercepting each request from the plugin app to the Android OS and vice versa to fool the OS into believing that the container issued the request. Anti-virtualization techniques aim to detect whether the app is executed within these virtual environments [35, 62, 84, 94, 97]. This could be done in different ways, for instance, by verifying its own UID, the number of running processes, or the object instance of the Android API clients.

Network artifact detection. This type of evasive control aims to inspect the network interfaces to detect artifacts, such as unusual interface names or ADB connected over the network. Moreover, since sandboxes often intercept and analyze the app's network traffic to understand its behavior, evasive apps may also check for the presence of VPNs or proxies.

3.2 APK tampering verification

Anti-tampering techniques detect any modification on the original app during its execution [29, 66]. If modifications are detected, the app can take evasive actions, such as turning off certain features or terminating its execution.

Signature checking. As Section 2 explains, APKs are digitally signed. This control checks if the certificate is the expected one.

Code integrity. These checks verify whether some code or resource has been tampered with by computing its signature at runtime and comparing it with pre-computed and hardcoded values.

Installer verification. Since API level 5, the Android Package Manager stores information on which 'installer' app (e.g., Google Play Store or Samsung Store) was used to install any given app. These evasive check retrieves the package name of the installer app to verify whether the app has been installed from the expected app store. Tampered apps are more likely to be distributed on unofficial app stores that differ from the original. Moreover, an APK can also be downloaded directly from a website, and thus, in this case, the installer app can be a browser or a file manager.

3.3 High-level verification

SafetyNet attestation & Integrity API. SafetyNet [51] is a platform security service offered by Google that provides a set of APIs to help protect apps against security threats, such as device tampering and other potentially harmful apps. For instance, to verify the integrity of a device, an app leverages the Attestation API by invoking the `attest` method of the SafetyNet client, while to check if malicious apps are installed on the device, an app can invoke the `listHarmfulApps` API.

Starting January 2023, the SafetyNet attestation is deprecated and replaced by the Play Integrity API [50]. It offers an enhanced security mechanism that verifies the app's integrity to defend against tampering and redistribution of your app and the environment in which it is running. Also, it consolidates multiple integrity offerings (including the ones offered by SafetyNet) under a single API.

Human Interaction. Even sophisticated Android malware sandboxes often neglect to mimic realistic user behavior and interaction. In 2022, Kondracki et al. [58] have shown how user-related artifacts (e.g., number of photos and songs, list of contacts) can be abused to distinguish an actual device from a sandbox environment.

3.4 Behaviors and Methods

All the evasive behaviors listed above can be implemented in different ways. In the rest of the paper, we assign a unique identifier to each method, which consists of the concatenations of three strings: the behavior, the method, and the type of control. For instance, the ROOT-SU-FILE method denotes a *root detection* evasive behavior, which aims to verify the presence of the `su` binary (method), and achieves that by looking at the *file* (type of control). Due to space constraints, we cannot include a complete description of all the 97 unique techniques we implemented in the paper. The interested reader can find all details in our Github repository [10]. The repository also contains Android-Al-Khaser, an Android app that implements a proof-of-concept of each evasive technique.

Finally, it is important to notice that while, in certain cases, we can detect the presence of a given evasive method used by an app, in other cases, an app can collect some general information that can be used internally to implement the evasive control. We call the first type a *Direct* evasion technique (DET) and the second an *indirect* one (IET). For instance, Magisk [19] is a famous open-source app to customize Android, which requires root access. To verify if this app is installed, a developer can interact with the `getPackageInfo` or the `getInstalledApplications` methods of the `PackageManager`. The former accepts the package name of the target app, while the second does not take any argument and returns a list of *all* apps installed for the current user. Thus, if a sample invokes the `getPackageInfo` method with the `com.topjohnwu.magisk` argument, we can flag it as a direct implementation of the root detection

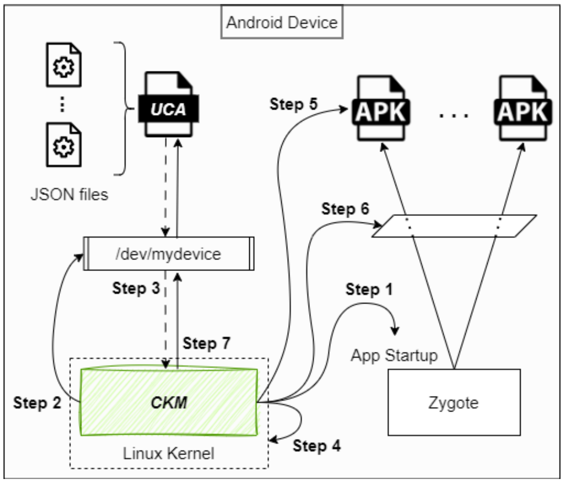

Figure 1: Sandbox overview

evasive method. However, a sample can also retrieve the entire list of apps by invoking `getInstalledApplications` and then look for the Magisk package name in several stealthy ways (e.g., by comparing the hash of each name). Hence, in this case, we can not be sure that the app is trying to evade the analysis, but we can still report that it collects specific information that can be used to implement evasive control in an *indirect* way.

4 *DROIDDUNGEON*

Figure 1 presents an overview of *DroidDungeon*. We implemented the monitoring and anti-evasion capabilities of the sandbox in a custom kernel module (from now on, *CKM*). This choice is less invasive than other techniques commonly used in malware analysis – such as userspace instrumentation, as pointed out by previous work [69, 88] – resulting in stealthier and sounder analysis. Precisely, *CKM* can monitor system and app *events* (i.e., Android APIs, library functions, and system calls), enforcing the anti-evasion criterion when necessary. Alongside the *CKM*, *DroidDungeon* also ships a userspace companion app (*UCA*), which provides configuration directives to the kernel module through a special device file (`/dev/mydevice` in the figure). These directives include the list of Android APIs and syscalls to monitor and the anti-evasion techniques to enable, which the analyst can tweak by providing configuration files to the *UCA*.

At first, the kernel module is automatically loaded during the device boot routine but remains dormant, waiting for the *UCA* to start (Step 1 of Figure 1). *CKM* detects this event by monitoring the `prctl` syscall that the Android operating system uses to name the main process of an app when it starts. At this point, the kernel module finds the Zygote's PID by iterating over the list of all running processes, and it creates a special device file (Step 2) to communicate with the *UCA* (Step 3).

The app then sends the categories of syscall that the module needs to hook[2] (Step 4), and the Android APIs to monitor (Step 5). The kernel module leverages the kernel *tracepoint*, *k(ret)probe*, and *u(ret)probe* subsystems[3]. In particular, *CKM* sets one tracepoint to tap the kernel's syscall trap handler that parses the parameters of each syscall invoked on the system, and one k(ret)probe or u(ret)probe for each specific syscall and Android API that logs the event and implements the relevant anti-evasion tricks.

During the module initialization phase, the *UCA* also sends the memory addresses of the `android.os.Build` Java class. This class is initialized during the device's start-up in the Zygote process, and some of its static fields are known to be exploited for evasion. Since in Android each app inherits its address space from Zygote, virtually all apps can access such valuable information, which *CKM* needs to modify as an anti-evasion countermeasure (Step 6)[4].

The module setup phase ends when the *UCA* sends the *init* command. At this point, the kernel module works event-driven, responding to the events triggering the registered tracepoints and probes and sending the corresponding log entries to the companion apps through the special device (Step 7).

4.1 *CustomKernelModule*: Implementation Details, Technical Challenges & Solutions

The *CKM* is a loadable Linux kernel module written in 11,842 lines of C code. It can intercept all the system calls, kernel functions, and compiled userspace code in ELF or OAT format and parse and modify native (C/C++) and Java objects. *CKM* monitors only the apps installed after the companion one by filtering out all the events of processes with a UID higher than the *UCA*. This prevents *CKM* from interfering with critical system components, which would make the operating system unstable. Last, *CKM* sends a log of all relevant recorded events to the *UCA* through the device file by using a protocol based on the eXternal Data Representation (XDR) standard [57].

Tracepoints and probes. To register tracepoints, kernel, and user probes, the *CKM* uses respectively the `register_trace_sys_enter`, `register_kprobe`, and `uprobe_register` standard Linux kernel functions. While `register_kprobe` and `uprobe_register` can be used by any kernel module, the Linux kernel prevents an external module from registering such a tracepoint. We circumvented this limitation by patching the Android kernel to export tracepoint-related functions and make them externally visible. This tweak consists of only two lines of C code (Listing 1 in Appendix), making it, in fact, extremely portable between the various kernel and Android versions.

Unlike tracepoints, whose targets are fixed and defined at compilation time, the *CKM* has to specify the targets of the kernel and user probes at runtime. The former can be set by specifying the kernel symbol name of the desired function to hook. The procedure is more complicated for the latter, as the `uprobe_register` function

[2]To give an idea, the syscall *connect* belongs to the network category. Table 2 in the Appendix recaps the list of syscalls for each category.

[3]These are built-in tracing mechanisms provided by the Linux kernel. Tracepoints and K(ret)probes allow one to register callbacks triggered any time a specific kernel function is executed. The callback has complete control over the calling process, including inspecting and modifying CPU registers and memory. While tracepoints can only be declared at compile time, k(ret)probes can be defined at runtime and allow tracing a specific function's beginning and end. U(ret)probes are the userspace equivalent to k(ret)probes and can be placed on any ELF or OAT file a process loads.

[4]In theory, it is possible to find the address of the `Build` class directly from kernel space by introspecting any app's address space. We opted to offload this task to the userspace app instead. Obtaining the class address is, in fact, much easier by taking advantage of the managed ART runtime from the app than it is from kernel space.

requires a specific offset of the ELF file at which to add the probe. We offloaded the offset calculation to the *UCA*, which delivers the binary file paths (i.e., the path of the ELF or OAT file) and the offset of each API to hook through the communication device.

Parsing probe parameters. To monitor execution events and implement anti-evasion measures, *CKM* needs to interpret the parameters the analyzed apps employes to invoke syscalls and APIs. The first step of this process consists in recovering the parameters' values according to the architecture calling convention. While implementing the parsing routines, we discovered that the OAT files adopt undocumented (at least to our knowledge) calling conventions on the x86 and x86-64 architectures. In particular, the first three parameters must be passed through registers on x86 or the first five in the case of x86-64, with the remainder on the stack. Moreover, Java instance methods have a hidden parameter pointing to the object on which the method is called (referred to as *this* in Java). Once the parameters are retrieved, *CKM* handles them according to the related event's semantics. For instance, to monitor the write syscall, *CKM* resolves the first parameter to the path of the output file and copies the data written by de-referencing the pointer provided as the second parameter.

Java object parser. While parsing syscall parameters is relatively easy, handling the parameters provided to Android APIs proved to be a more complex task requiring knowledge about how the ART runtime stores Java objects in memory. In particular, each Java entity can be classified into four categories according to its type, namely Primitive, Array, String, and Complex [24]. While Primitive entities store basic data types (e.g., bool, char, int), all the other categories represent Java *objects*, i.e., structured data types that inherit their structure from the `java.lang.Object` class. The Object class has two members: a four bytes pointer to a `java.lang.Class` object and a four bytes hash.

The object's Class defines the size of the object instance, its superclass, and the list of fields as an array of `Art_Field` objects. Each `Art_Field` specifies a pointer to the declaring class of the field (from which the field name can be retrieved) and its offset (in byte) from the beginning of the Complex object. Thus, by parsing the array of `Art_Fields`, we can decode the Complex objects as a set of fundamental object instances. It is also important to note that a field of a Complex object could, in turn, be another Complex object or an Array of Complex objects. This makes parsing Java objects a recursive procedure that ends when a Primitive or a String object is found. To our knowledge, *CKM* is the first in-kernel runtime Java object parser and modifier.

Custom stack unwind. In *DroidDungeon*, we have mainly adopted a "fake response" approach by modifying the output values of several system calls, Android APIs, and library functions when invoked by the app under analysis, mimicking the behavior they would show on an actual device. However, indiscriminately providing fake responses to all invocations of these functions can be detrimental. For instance, several evasion techniques consist in scouting the file system in search of QEMU-specific device files. A naive approach to counter these stratagems would be to counterfeit the result of all file system syscalls that could reveal their existence. In doing so, however, we would hinder the intended uses of such devices that the Android framework opens any time a new app starts, resulting in an unrecoverable exception. To handle this and other cases, we developed a more fine-grained technique to assess a call's purpose by considering its *provenance*. *CKM* tampers its outputs only if the call originates (directly or indirectly) from the app's code. On the other hand, if the call does not originate from the app's code, it must be part of the intended operations performed by the Android framework, and *CKM* should not meddle in its execution.

To assess a call's provenance, *CKM* unwinds the call stack, reconstructs the list of function calls, and looks for those that belong to the app's code. Reconstructing the calling function list means retrieving the list of return addresses from the stack. The first element in this list (i.e., the caller's address of the hooked function) is the value in the stack immediately before the area pointed by the stack pointer. The following elements in the list can be computed iteratively by reconstructing each caller's stack frame and retrieving the respective return address. At any given step of the stack unwind procedure, *CKM* computes the next frame pointer (*FP*) as:

$$FP_{(n+1)} = FP_n - FS_n - (sizeof(void*) * SA_n)$$

where FS_n is the frame size of the n-th function in the call stack (i.e., the sum of the sizes of all stack variables of that function), and SA_n is the number of its parameters passed on the stack. Since neither *FS* nor *SA* can be easily retrieved at runtime, we opted for computing them *a-priori* for every symbol of every library in the Android framework. To this end, we developed a Ghidra [17] script and a Python program based on *oatdump* that computes each function's frame size and stack parameters for Android ELF libraries and OAT files, respectively. The value calculated for each function in the Android libraries is then embedded in the *CKM* at build time.

The stack unwind procedure ends whenever it encounters an address that belongs to the app's code or an invalid address. In the first case, the module verifies which method or API the app invokes to determine whether to enforce the anti-evasion policy. When, instead, the unwinding procedure reaches an invalid address, *CKM* infers that the event was due to an Android standard routine (e.g., app start-up) and does not enforce the anti-evasion criterion.

Notice that this approach works even against reflection. Invoking a framework method through the reflection only adds a few function calls between the caller and the callee, which our stack unwind routine can handle seamlessly.

To our knowledge, *DroidDungeon* is the first analysis system that performs *call provenance* test through an in-kernel stack unwinder.

JIT & stack unwind. In Android, the `libart.so` library enforces the JIT compilation and contains the *jit functions* to manage all DEX instructions (i.e., `nterp_op_<inst_name>` functions). For instance, `nterp_op_return_void` and `nterp_op_return_object`, respectively, handle the return statement of a Java (`void`) procedure or a function that returns an `object`. Android interprets the non-OAT-compiled DEX code to build a chain of equivalent jit functions.

We discovered that also these functions do not respect the standard calling convention: the jit functions do not have a prologue and epilogue, ending the procedure with a `JMP` instruction instead. Moreover, the return address – a pointer to the following jit function in the chain – is computed at runtime. Thus, the stack unwinding routine does not apply to this scenario (except for the `nterp_op_return*` functions, which have an epilogue and end with the standard `RET` statement returning to the pointer stored on the stack).

The idea we had to manage JIT-compiled functions in the stack unwind procedure is that once the execution reaches the jit function chain, the stack frame does not changes until the return is reached. It is worth noticing that a JIT-compiled function always ends with a return statement regardless of the first DEX instruction. Also, during the stack unwinding procedure, *CKM* reaches a JIT-compiled function if and only if it invoked another API or a native method, i.e., if it came from one of the `nterp_op_invoke_*` jit functions. Thus, once one of these functions is reached, *CKM* can consider as if the return jit function had been invoked: it adds the return's stack frame size to the SP, retrieving the caller's return address from the stack as usual.

4.2 *UserspaceCompanionApp*

The *UCA* is a regular Android app written in about 1,620 lines of Java and 1,283 lines of C code. It supports the *CKM* in Steps 2 and 7 of Figure 1. The JSON configuration files specify the set of functions the *CKM* has to hook and which anti-evasion techniques need to be enforced. Contrary to the kernel probe, which can be registered by specifying only the function name, *UCA* retrieves the offset of the target userspace functions from the ELF or OAT files, which *CKM* needs to register the uprobes. In particular, the app leverages the `readelf` [21] utility for the ELF files, while it uses `oatdump` for the OATs. Finally, *UCA* listens on the device file to log the intercepted events into a regular file.

4.3 Anti-evasive Policy

The *DroidDungeon* design allowed us to limit the artifacts we need to manipulate to only the emulator and network categories (refer to Section 3). Specifically, once the kernel module receives the init command, it modifies the properties of the network interfaces (e.g., their names) to make them appear similar to the ones in an actual device. Also, it installs probes to monitor and tamper with file-related operations (Step 4 of Figure 1) and manipulate high-level Android APIs (Step 5), including changes to system properties (e.g., `ro.hardware`) and the simulation of real sensors.

We recall that some static fields of the `Build` class can be exploited for evasion, and *CKM* modifies them each time a new app starts (Step 6).

Finally, the device has a user logged in to an actual Google account, and the file system is populated with a collection of documents, images, and other common files to resemble a legitimate smartphone so that malware may identify it as a more valuable target [67].

5 EXPERIMENTAL SETUP

Dataset. To perform our analysis, we built a comprehensive dataset of Android apps divided into malware and goodware samples. We collected malicious apps from the VirusTotal (VT) feed [92], a real-time stream of JSON-encoded reports containing the analysis results for each app submitted to VT. We wanted our dataset to be diverse regarding the number of families and balanced so that every malware family is well-represented. For this reason, we monitored the feed from September 2022 to April 2023. We only retained the Android apps identified as malicious by at least five engines and fed them to the AVClass2 malware labeling tool [82], which outputs the most likely family name for the sample. At the end of the collection, we ended up with *20,556* malicious apps, uniformly distributed over 200 families.

For goodware, we collected the package names of the 500 most downloaded free apps for each of the 50 official Google Play Store categories in April 2023. We extracted this information using Google Play Scraper [8] and downloaded the apps thanks to apkeep [12] directly from the Google Play store. In total, we collected *21,154* unique samples.

Runtime Environment. We used *DroidDungeon* to conduct the first analysis of public evasion techniques in goodware and malware; thus, we implemented it in an Android emulator. Since February 2020, Google has introduced support for running ARM binaries on x86 (Android 9) and x86-64 (Android 11) system images. However, starting from version 12, Android emulators can execute only 64-bit binaries. Thus, to execute ARM and Intel 32- and 64-bit-based Android apps, our analysis system comprises two emulators based on Android versions 12 (API level 31) and 11 (API level 30). Before running an app, we check if it only has 32-bit libraries and choose the appropriate emulator.

Moreover, because *DroidDungeon* can not hook DEX code, we must ensure that every Android Java function is OAT compiled. Thus, we replaced all the OAT framework files in our Android emulators' official `system.img`. We exploited the AOSP build tools to unpack the image, OAT-compile all the Java APIs, and repack it in a new one. It is worth noticing that this step is not straightforward: Android put in place several techniques to validate the integrity of the framework files; thus, we took into account every check. For instance, since version 8, Android performs the verified boot process [11], which assures the integrity of the framework software, also verifying the `system.img`. Thus, we had to recreate a valid `vbmeta.img`, namely, the data structure contains all the metadata for the verified boot process. In this way, we deployed *DroidDungeon* with the complete support of the Google Play services.

All emulators run on a dual-core 2.10 GHz x86-64 processor, 2 GB of RAM, 16 GB of internal storage, and an 8 GB emulated SD Card. Finally, we manually tested twelve apps (both malicious and benign) that we knew how they worked and found no user experience problems or malfunctions.

Red and Blue runs. The concept of red pill and blue pill in evasive malware [37, 71] refers to the movie "The Matrix", where the protagonist is offered a choice between a red pill that reveals the true nature of reality and a blue pill that keeps him in a simulated world. Thus, a red pill is an evasive technique, while its blue pill is a corresponding defensive technique to counteract the red one.

In this work, we also wanted to study how much the executions vary when *DroidDungeon* gives blue pills compared to when it does not provide them. Hence, every app in our dataset is executed twice. In the first run (*BlueRun*), *DroidDungeon* hides the underlying emulator by administering blue pills (i.e., it enforces the anti-evasion criterion). In contrast, in the second run (*RedRun*), the app is executed without modifying the emulator's behavior.

In every app execution, *DroidDungeon* stimulates the app user interface for 4 mins to increase its code coverage and simulate a real user with ARES [77]. This black-box tool uses Deep Reinforcement Learning to test and explore Android apps. Moreover, we modified

Table 1: Main differencences of evasive technique usage for malware w.r.t. goodware.

	Evasive Technique	Malware	Goodware	Malware - Goodware
DET	INSTALL-SOURCE-API	43.3%	88.7%	−45.4%
DET	VIRT-UND_PERMS-API	35.9%	76.3%	−40.4%
DET	ROOT-SU-FILE	18.3%	56.1%	−37.8%
DET	EMU-SYSTEM-API	44.1%	19.8%	+24.3%
DET	EMU-QEMU-FILE	3.9%	1.2%	+2.7%
DET	EMU-KNOWN_EMU-FILE	3.2%	1.1%	+2.1%
IET	SIGNATURE-APP-APP_INFO	25.6%	76.2%	−50.6%
IET	NET-SSL_PINNING-API	54.6%	78.1%	−23.5%
IET	VIRT-FAKE_COMP-API	31.2%	47.1%	−15.9%
IET	HOOK-PROC_ART-MAPS	26.2%	13.0%	+13.2%
IET	HOOK/ROOT-APPS-INST_APPS	19.4%	6.5%	+12.9%
IET	NET-INTERFACE-NF	14.7%	5.9%	+8.8%

ARES to perform the same sequence of user clicks for every tested app to compare the two execution traces.

Preliminary results. We were able to execute correctly the 93% of malware (19,090/*20,556*) and the 99% of goodware (21,081/*21,154*). For the failed apps, we were not able to install them because the signature was not valid or the APK file itself was corrupted.

6 RESULTS OF THE MEASUREMENT

This section discusses the results of the analysis we conducted over the malware and the goodware datasets. Starting from the app's execution traces, we developed a post-analysis routine that identifies DETs and IETs by looking at the list of events that occurred after the first access to the `base.apk` file (which, as explained by Ruggia et al. [78], signals the start-up of an Android app). When reporting the results of evasive behaviors and methods, we will use the unique identifier (in uppercase) we introduced in Section 3.4.

6.1 Prevalence

DETs and IETs usage. 90.8% of goodware and 68.5% of malware implement at least one DET evasive technique, while about 90% in both categories contain at least one IET. On average, the malware uses 2.1 unique DETs ($\sigma = 1.9$) and 12.8 ($\sigma = 6.7$) IETs, while goodware 3.4 ($\sigma = 2.2$) and 14.4 ($\sigma = 4.9$), respectively. Interestingly, goodware has almost twice as many DET controls as malicious samples on average. However, the sample that employs the maximum number of unique DETs (15) and IETs (39) controls is malicious, which is almost 15% higher than the maximum for goodware.

While the prevalence is high in both groups, there are important differences in the techniques adopted by malware and goodware. Table 1 reports the three DETs and IETs that differ the most among the two groups. In particular, it shows the percentage of malware and goodware that implement a specific technique w.r.t. all the dataset apps. A negative value in the rightmost column means that such an evasive check is implemented more often in goodware, while a positive means is more prevalent in the malware dataset.

First, more than 88% of goodware verifies how the app was installed or updated by invoking the `getInstallSourceInfo` method of the `PackageManager` (`INSTALL-SOURCE-API`), clearly showing how developers care if the app comes from the expected store.

Interestingly, 76.3% of goodware verifies the environment in which they are executed by checking if permissions not declared on the Manifest are granted to the app (`VIRT-UND PERMS-API`). It is a common technique to detect whether the app is running in an app-level virtual environment [61, 90] because container apps have to declare all possible permission to manage with a generic plugin app. However, we have investigated this further, discovering that most goodware samples import third-party libraries for analytics and monetization (e.g., [13–15, 18]), which check the granted permissions at runtime to extract as much information as possible.

The third DET control is related to root detection: 56.1% of benign apps check the presence of the `su` file (`ROOT-SU-FILE`).

On the other hand, in malware, DETs controls are related to detecting the emulated environment or an analysis system. In particular, 44% of malware retrieves and checks Android system properties, such as the device or the subscriber id, by interacting with Android managers (`EMU-SYSTEM-API`). In addition, malware samples are more prone to searches for well-known emulator artifacts (e.g., `EMU-QEMU-FILE` and `EMU-KNOWN_EMU-FILE`) and network proxy apps.

Concerning the IETs, goodware often retrieves information about the certificate used to sign the APK by querying the package manager (`SIGNATURE-APP-APP_INFO`), performs SSL pinning (`NET-SSL-_PINNING-API`), and checks for artifacts in the process components through the Android APIs (`VIRT-FAKE_COMP-API`).

Conversely, the main IET controls in malware are related to anti-hooking, root checks, and network artifact detection. First, malware verifies process artifacts by checking the content of the `maps` file in the proc file system (`HOOK-PROC_ART-MAPS`). Moreover, almost 20% retrieves the list of all installed apps on the device and then perform some checks over it (i.e., `HOOK/ROOT-APPS-INST_APPS`). This technique can be exploited for detecting hooking (e.g., Xposed) and rooting (e.g., Magisk) apps. Even if there is proper permission for doing it, Ruggia et al. [78] demonstrated that there are tricks a malicious app can leverage to bypass this protection mechanism. For instance, restrictions are not applied to apps targeting API level 30 or lower, which can retrieve the metadata of any app in the system. Interestingly, 90% of malware on average targets an older API (<= 30), and almost 3% requests the QUERY_ALL_PACKAGES permission. The picture is different for goodware: only 14% targets an API before level 30. Compared to malware, it is a small value, but, in absolute terms, it is unusual that benign goodware apps do not update the target API as guidelines.

Last, malware verifies network interface properties by using native functions, such as `getifaddrs` (`NET-INTERFACE-NF`), to figure out if the device uses a VPN.

SafetyNet & Integrity APIs. During our analysis, we also investigated if and how goodware and malware use one of the SafetyNet APIs or the Integrity API. Since these technologies are not open-source, we manually analyzed the Android framework to figure out their inner workings.

For **SafetyNet**, we intercept its binder request, the typical Android inter-process communication, and remote method invocation technique. Table 3 in the Appendix reports the entire mapping between binder methods id of the SafetyNet client to its "high-level" security check. Both goodware (35.7%) and malware (30.4%) interact with at least one of the security services SafetyNet offers.

One of the services offered by SafetyNet is Verify Apps. This service is unavailable by default, but an app can ask the user to activate it through the `enableVerifyApps` method. Then, an app can check if it has been enabled using the `isVerifyAppsEnabled` method. Once the service is enabled, users can check the list of harmful apps by invoking the `listHarmfulApps` method. Among samples that use SafetyNet, over 95% of malware and 90.6% of goodware samples communicate with the Verify Apps service, and all invoke the `isVerifyAppsEnabled`. However, surprisingly, just one goodware invokes the `listHarmfulApps` method.

Also, about one fifth of malware (14%) and goodware (19%) use the Safe Browsing API: they invoke the `loadUri` method to check whether a URI is linked to a well-known threat. However, this measurement is limited: we cannot distinguish whether it is an explicit invocation because the Android WebView automatically invokes this mechanism. Starting in April 2018, WebView supports the Safe Browsing feature by default [47, 49], automatically verifying the URI through this method.

Regarding the Attestation API, we observed only 0.8% of goodware samples and just *one* malware that leveraged it to verify the legitimacy of the execution environment. It is worth noticing that, contrary to other SafetyNet services, the Attestation API requires an app to have a registered and valid API key on its Google website; however, this malicious sample demonstrates that the attackers can abuse "benign" security mechanisms. Moreover, our data confirms the findings of Ibrahim et al. [53], showing that legitimate apps do not properly use SafetyNet services that would significantly improve their security posture.

To monitor the **Play Integrity**, we intercept the intents used to communicate with this component. Given that this service is very recent (explicitly created to replace SafetyNet), we monitored only a negligible amount of goodware (0.11%) and no malware using this API. We argue that this new service will be harder to exploit by malicious actors because it is strictly tied to the Google Play Services and requires many verification steps.

Time-based Analysis. We normalized the execution time of each sample in a [0,100] range (as suggested by [63]), and then divided it in three-time slots: [0-10], [11-89], and [90-100]. Then, we tracked when the first and last evasive checks (both DETs and IETs) were performed. Moreover, we also considered the difference between the last and the first to examine whether checks are usually executed in quick succession or at different points in time. From a preliminary analysis, we noticed that there were no significant differences if we considered DET and IET separately; therefore, for ease of reading, we consider them together.

The KDE plot for malware and goodware are shown in Figure 2 in the Appendix due to space limit, although we report the relevant observations. The percentages of the first and last evasion techniques occurring during the first slot [0-10] of the execution are 63.6% and 10.2% for malware and 73.4% and 2.7% for goodware, respectively. Interestingly, most goodware performs evasive checks at the beginning of execution, even more than malware. Then, the last evasive technique falls in the last slot [90-100] for the 30% of malware and 39.3% of goodware.

Then, for each slot, we computed the percentage of how many times a specific control has been used. For goodware, the three most common evasive controls are `INSTALL-SOURCE-API`, `ROOT-SU-FILE`, and `VIRT-UND_PEMRS-API`, regardless of the time slot. Thus, benign samples verify from where they were installed and whether the `su` binary file is present or some non-declared permission is granted. Contrary, for malware, evasion checks depend on the timing. In the first time slot, malicious samples verify the `maps` file in the proc file system (HOOK-PROC_ART-MAPS) and from where they are installed (`INSTALL-SOURCE-API`). Then, regardless of the second and third slot, they look for Android emulator fingerprints (e.g., device ID) and network-related information (`NET-INTERFACE-API`, i.e., the name of the network interface).

Finally, we also investigated the order in which goodware and malware samples performs the evasive checks. Evasive goodware is predominantly characterized (54%) by controlling the installation source (INSTALL-SOURCE-API), while only 6% of evasive malware uses it as the first control. On the other hand, for about half of the malicious evasive samples, the first control is `HOOK-PROC_ART-MAPS`, followed by `EMU-SYSTEM-API` (12.5%) and `NET-LISTENER-API` (10%). The order in which apps utilize different strategies is crucial because researchers need to appropriately mitigate them to avoid limiting the results to those evasive techniques used first.

Evasive among families/categories. We assessed if the goodware category (e.g., banking or game) or malware family affects the overall evasiveness of a sample. It is reasonable to assume that benign samples are more prone to implement evasive techniques if they manage sensitive user information (e.g., banking and finance).

We found that 195/200 (97.5%) malware families contain at least one sample that uses a DET, while if we include the IETs, the number of families goes up to 199/200. Also, for 22/200 (11%) malware families, *all* the samples in the family contain at least one DET. These numbers indicate that it is crucial to consider this phenomenon in the dynamic analysis of Android malware. On the other hand, all goodware categories contain at least one evasive sample, but none have all samples with at least one DET technique. Moreover, we measured the variation of the number of evasive techniques in malware w.r.t. goodware. On average, the standard deviation of malware is almost four times w.r.t. goodware; namely, the number of controls carried out by malware apps depends on their family. In contrast, the app category only affects a small number of evasive checks in goodware.

We also assessed the 'evasiveness' of a family by counting the number of evasive techniques for each sample in the family. On average, the most evasive malware family is *loead*, whose samples implement, on average, 16.3 evasive controls. This is followed by *fydad* (14.7), *beitaad* (11.2), and *snaptube* (11.1). In the benign dataset, *Finance*, *Entertainment* (e.g., Netflix and Twitch apps), and *Shopping* are the most evasive goodware categories (with an average of more than 6 evasive controls per sample).

Finally, we measured how many categories/families use a specific evasive technique. For goodware, 21 DETs and IETs are implemented by more than 80% of the app categories, while this number decreases to 12 for malware families. The most widespread for goodware (almost all goodware categories) are verifying process artifacts, checking the APK signature or whether permissions not declared in the Manifest are granted. On the other hand, the most common for malware are emulator (e.g., `EMU-SYSTEM-PROPS`) and hook detection checks (e.g., `HOOK-FRIDA-FILE`). Interestingly, some checks (e.g., `EMU-KNOWN_EMU-FILE` and `HOOK-PROC_ART-MAPS`) are more spread in goodware categories (86%) w.r.t. malware families (32%), even if their overall occurrence is higher in malicious apps (e.g., EMU-KNOWN_EMU-FILE occurs in the 4.3% of malware and only 1.6% of goodware).

RQ1. Our experiments show that goodware and malware use evasive techniques for different purposes. For instance, about 70% of evasive malware implement at least one environment verification control related to the emulator detection, while only one third of evasive goodware does it. Contrarily, the most common environment verifications for goodware are app-level virtual environment (81%) and root checks (61%), but they account for respectively only 43.4% and 33% of malicious samples. Goodware is also more prone to implement APK tampering verification; e.g., more than 82% of all benign samples perform signature verification controls.

We also observed that malware tends to rely on IETs techniques more often than on DETs. For instance, for root detection, goodware verifies if the su binary exists, while malware interacts with the package manager to retrieve the list of all installed apps.

Finally, our results highlight how malware families affect the type of evasive controls, while goodware apps tend to employ the same evasive techniques regardless of their categories. However, benign apps implement a heterogeneous set of controls for each category because the variation of the evasive controls in every category is higher than in the malware family. Each category has samples with several and no evasive checks, showing how evasive controls are implemented, especially by the most popular apps.

6.2 Evasive w.r.t. Packers & Protectors

We now look at the relationship between evasive behaviors and the presence of packing schemes and software protectors. We used APKiD [76] to determine whether a sample uses packing or protection techniques. On the goodware dataset, APKiD could not recognize any packer or protector. Conversely, APKiD recognizes 24 different packers in 11.4% of the malware samples. As reported in Table 4, the Jiagu packer is the most common in our dataset and was detected on 43.0% of the packed samples. Tencent, Baidu, and MultidexPacker are the following most prevalent packers, accounting respectively for 9.9%, 5.8%, and 5.3%; the remaining 10/24 packers account for less than 1%. On the other hand, only 0.2% of the malware apps are protected by two different protectors: Virbox and CNProtect. Given the negligible number of protected apps, our analysis is focused only on packers with a non-negligible prevalence (> 1%).

In this context, our goal is to understand whether any difference exists in the adoption of evasive behaviors between packed and non-packed samples. We started by measuring the proportion of apps that implement at least an evasive check, and found a similar prevalence between the two classes - respectively 82% and 89% for packed and non-packed apps.

We further investigated whether specific evasive controls are more characteristic of the packed apps compared to the non-packed ones. We found that the most used evasive techniques in packed malware are SIGNATURE-ZIP-FILE, HOOK-PROC_ART-MAPS, and HOOK-FRIDA-FILE. On average, the former IET technique (i.e., opening the base.apk file through the java.util.zip.ZipFile Java utility) is used by more than 65% of packed apps, with a peak of more than 90% for Ijiami and ApkEncryptor. On the contrary, such a technique is used by 26% of the samples when considering the collection of non-packed malware and less than 20% for goodware. From an evasive point of view, this operation is useful to access and read specific files in the APK, to perform signature checks or code integrity. However, packers also leverage this mechanism to unpack the APK file (without unzipping inside the disk) and load resources or encrypted code.

Nine packers cause the apps to open the fd/*, task/*, or maps files under the proc file system (i.e., HOOK-PROC_ART-MAPS, HOOK-FRIDA-FILE) more frequent compared to non-packed ones. In particular, these operations occurred more than 60% of packed apps (with a maximum of 81% for Tencent, Baidu, and Bancled), while this value decreases to 25% for non-packed malware. These IET techniques verify the process artifacts to identify changes and hook mechanisms, such as the instrumentations injected by Frida.

We finally investigated how varied the evasive controls are for each packing software. We detected that APKs packed with Jiagu implement the highest number of unique evasive controls (56 over 97 techniques identified in this study [10]), closely followed by the apps packed with ApkEncryptor and DexProtector, on which we respectively identified the presence of 55 and 51 different evasive techniques. On the other side, we could only measure 9 distinct evasive checks on samples packed with Bangcle. In a deeper investigation, we found that for some packers all the samples implement specific evasive controls: SIGNATURE-ZIP-FILE always characterizes samples packed with MultidexPacker, while those packed with AppSealing and ChronClickers always control for EMU-SYSTEM-API and HOOK-PROC_ART-MAPS. Overall, we measured that although some techniques are more prevalent in packed samples or in specific packing schemes, all evasive behaviors used in packed samples also exist in non-packed ones and vice versa. This highlights that, in the considered dataset, evasive techniques are not strictly related to the usage of a packer.

RQ2. A comparable ratio (82% and 89%) of packed and non-packed samples implements at least an evasive technique. Similarly, packed and non-packed APKs implement, on average, 5.1 and 8.7 evasive checks, respectively. Nevertheless, some behaviors such as SIGNATURE-ZIP-FILE, HOOK-PROC_ART-MAPS, and HOOK-FRIDA-FILE are more widespread in packed samples than in non-packed ones. Moreover, samples packed with specific packers always implement a subset of evasive controls: for instance, samples from AppSealing and ChronClickers always check Android system properties (EMU-SYSTEM-API) and process memory artifacts (HOOK-PROC_ART-MAPS), which are observed on less than 40% of non-packed samples. We could not find evasive techniques exclusively employed by packed samples or specific packing routines.

6.3 *BlueRun* vs. *RedRun*

In our final experiment, we measured the difference between the execution traces: the *BlueRun* in which our sandbox mitigates the anti-evasion mechanisms, and the *RedRun* in which it does not. The expectation is that an app employs evasive checks to *avoid* executing a particular piece of code. However, it is not trivial to establish a methodology to compare two traces and find this difference. The reason is that we cannot make assumptions about code that does and does not execute. Other works [22, 86, 93] have faced a similar problem; however, they addressed it for their specific use case: [86] only compared the number of file operations, while [22, 93] examined different forms of a call graph. Therefore, we measured the differences between the events in the execution traces by dividing them into 11 high-level categories related to the workings of Android: 1) *Accessibility Service* (a11y), 2) *Broadcast Receivers* (BR), 3) *Command Line Interface commands* (CLI), 4) *Content Providers* (CP), 5) *Dangerous APIs* (DAPI), 6) *Dynamic Code Loading* (DCL), 7) *File System* (FS), 8) *Inter-Process Communication* (IPC), 9) *Network* (NET), 10) *Requests for Permissions* (PERM), and 11) *Systems Services* (SS).

To conduct this measurement correctly and verify whether a pair of events were the same, we had to post-process the traces to remove execution-specific values (e.g., a path with UID or temporary network tokens). We were guided by some works [26, 36] where, for example, the authors generated ML features related to file system activity by removing the file name and just considering the path plus the file extension; or else, when dealing with a URL, they were considering protocol, domain, and port.

In addition, there are some important factors that we considered. First, given that we ran our experiments on Android emulators, all the blue pills *DroidDungeon* inject are designed to hide our specific environment; thus, there may be cases where the sample does evasive checks, but we do not provide such blue pills because our environment does not need them. For example, the `su` executable is not present on our emulator, so if an app checks for the presence of this file, it will not find it in both executions. On the other hand, if an app checks for the `/dev/goldfish_pipe` file (the presence of this device reveals an emulator) in *RedRun* it finds it, while in *BlueRun* it does not. For this reason, the numbers we will report below must be considered a lower bound. Second, we considered the events of the two traces divided into categories as elements of two ordered sets that we will call Red (R) and Blue (B) for brevity. $B_{<cat>}$ denotes the set of events in a specific category (e.g., B_{NET} is the collection of NETwork events of *BlueRun*). Of these sets, we calculated union, intersection, the two differences, the various cardinalities, Jaccard Index, and finally aggregated by category, checking in percentages what the significant differences were. We also computed the percentage number of times an event occurred in B and not R and vice versa. Table 5 in the Appendix summarizes these results. For space and ease of reading, we only report the significant differences for evasive apps between B and R traces.

Moreover, although these measurements are only meaningful for the evasive samples, we also verified the non-evasive ones to double-check that our sandbox worked properly. Therefore, for all the numbers we reported, we have verified that the same trend does not occur for the non-evasive samples. During this inspection, we found that irrespective of the classes (malware/goodware) and category except for NET, for the non-evasive samples, the two traces are equal ($R_{<cat>} = B_{<cat>}$) in more than 90% of the samples, while this percentage varies considerably for evasive ones (between 30% and 96%). The only notable exceptions are the network traffic traces: they are very different even in non-evasive goodware samples ($R_{NET} = B_{NET}$ only in 46%). The main reason is related to ads and monetization libraries: at each execution, goodware renders different ads, which generates different network requests.

A closer look at the cases where $R_{SS} \subset B_{SS}$, i.e., when the two traces are not the same and the *RedRun* events are a subset of the *BlueRun* ones , allowed us to observe that almost 14% of malware does not create any Android Activity when they are executed in *RedRun*, but they do it when the anti-evasion criterion is enforced; this phenomenon occurs for less than 4% of goodware. In practical terms, it means that these samples did not show any GUI and finished execution when their evasive controls detected a potential analysis environment. Nevertheless, contrary to other systems (e.g., Windows [37, 63]), most evasive malware samples (86%) do not stop their executions if the anti-evasion criterion is not enforced, behaving like a legitimate Android app. Combining some observations in the DAPI, CP, and SS categories, we noticed two interesting behaviors that evasive malware exhibits when it thinks it is not under analysis. First, almost 10% more samples interact with the Captioning Manager (that contains methods to access and monitor preferred video captioning state), and most of them (70%) use it as an alternative way to get the user's properties, such as the preferred language. Second, 2% records the audio or video of the device for an unlimited period through methods of the `MediaRecorder` class.

Regarding the FS category, more than 17% of evasive goodware looks for the `su` binary in *BlueRun*, while they did not do it in *RedRun*. We observed that these goodware samples perform root checks only after verifying the presence of an emulator. Conversely, it happened for less than 3% of evasive malware, which, on the other hand, are more prone to change the mode bits of files in their private folders. For instance, the `fchmod` syscall occurred more than 20% of evasive malware in the B_{FS} trace w.r.t. the R_{FS} one. We manually investigated this latter fact and found that, in these cases, malware makes some files containing code, such as native libraries, writable to modify their content (e.g., the Grifthorse Trojan [5]). In this way, the sample will execute a different code at runtime compared to the statically available one in the APK. Finally, the CLI category is abused by 1.5% of evasive malware to retrieve system properties value through the `getprop` command or run `chmod` command ($<$ 0.1 for goodware), avoiding interacting with the Android APIs.

RQ3. Our experiments show that 14% of malicious and 4% of benign samples refrain from running in our analysis environment when we do not mitigate evasive controls. In other cases, malware hides techniques to obtain information about the device or records audio/video without the user's knowledge. It also overwrites portions of its code to perform different operations than those that could be observed by statically analyzing the APK file. Goodware, instead, tries to hide its search for the presence of a device with root.

7 RELATED WORK ON ANDROID SANDBOXES

According to [86], we grouped current Android sandboxes for malware analysis based on the technique on which they are based. There are currently no Android malware sandboxes based on app-level virtualization, so this solution was not considered.

Full-system emulation. In the last years, researchers proposed several sandboxes based on Android emulators. In 2012, Yan et al. developed DroidScope [96], a virtual machine introspect (VMI) system to monitor the activity of the malicious app in the Android emulator. In 2015, Tam et al. proposed CopperDroid [88], a sandbox built on QEMU to automatically perform out-of-the-box VMI-based dynamic analysis and reconstruct Android malware behaviors. In the same year, DroidBox [74] and CuckooDroid [60] have been proposed. The former is a custom Android OS version 4.1.1 variant that tracks and taints API calls. Alternatively, CuckooDroid is an extension of Cuckoo Sandbox [42] for automating the analysis of Android apps; it is based on the Xposed Framework to monitor API calls and provide blue pills to the target apps. Similarly to CuckooDroid, over the years, other researchers proposed hook-based sandboxes which rely on Xposed [30, 34, 43] or Frida [40]. In 2018, Liu et al. [59] proposed RealDrois, an emulator-based analysis system built by modifying the Android framework.

Android Container-Based Virtualization. Similarly to the desktop counterparts, the Android container-based virtualization is a lightweight in-kernel virtualization technique that creates an isolated (virtual) environment in the same Android device. However, Android container development has to overcome several challenges; in particular, mobile devices are not be designed for multiplexing hardware components (e.g., WiFi and Bluetooth).

In 2011, Andrus et al. proposed Cells [25], a virtualization architecture enabling multiple isolated virtual phones (VP) to run simultaneously on the same physical device. In 2015, Xu et al. developed Condroid [95], a lightweight Android virtualization solution based on container technology, which leverages both Linux namespaces and cgroups to create multiple VPs. In 2021, Song et al. proposed VPBox [86], an Android OS-level sandbox framework via container-based virtualization that overcame the limitations of the previous works, integrating with the principle of anti-evasion. In particular, VPBox offers complete device virtualization for all the device components (e.g., WiFi, Camera), minimizing the artifacts in the VPs, and it can customize the virtual device configurations (e.g., OS version) for each VP.

***DroidDungeon* vs. SOTA.** Emulators are programs that simulate the functionality of some hardware, thus providing great scalability and flexibility. For instance, the virtual environment can be restored to a clean snapshot in seconds. However, the anti-evasion criterion is demanded to the sandbox itself, which has to implement the "bypass" logic for each evasive technique. In particular, DroidScope, CopperDroid, and DroidBox do not enforce any detection resilience mechanism, while hook-based techniques can bypass only a subset of evasive controls, leaving several artifacts uncovered. For instance, Xposed is not designed to hook into lower-level system calls; hence an attacker can detect the emulator by making direct syscalls. *DroidDungeon* leverages the probe mechanism, which allows the hooking of user functions and system calls. Thus, we can provide more fine-grained analysis mechanisms that only hook the least possible set of functions. For instance, to identify the opening of a file, *DroidDungeon* hooks only file-related system calls, avoiding hooking all the high-level functions for both the Java and the native layers. Moreover, all the container-based virtualization techniques and framework-level modifications (e.g., RealDroid) heavily modify the Android kernel and OS layers, which can make integration with system updates challenging (no maintainability criterion); in addition, the former techniques share the assumption to be executed on an actual phone affecting scalability.

It is worth noting that the main goal of container-based virtualization is to create an isolated environment. Still, more is needed to provide a suitable technique for monitoring and analyzing the app's behavior. Contrary, *DroidDungeon* can be deployed in both an emulated and actual device. In the former case, it has to enforce the anti-evasion criterion, while the second one leverages the underlying actual device to bypass evasive checks, behaving like a virtual phone. Moreover, we developed a fully separate kernel module, which can be easily integrated with newer Android versions.

Table 6 in the Appendix compares *DroidDungeon* and the other state-of-the-art solutions based on the anti-evasion, scalability, and maintainability criteria. At the time of writing, our solution remains the best tool to analyze Android malware dynamically.

8 LIMITATIONS AND CONCLUSION

Limitations. First, the probe mechanisms cannot measure direct memory access or reading Java object fields. For instance, simple evasive checks aim to verify the fields of the `Build` class. We cannot measure these events even if *DroidDungeon* can mitigate them by updating the `Build` class field values when a new app starts.

Second, when a probe is placed in a userspace program, the instruction at the probed location is overwritten by a jump to the handler routine. This mechanism introduces artifacts into the memory of the probed functions that an attacker could exploit to detect the sandbox.

Third, Garfinkel et al. [45] demonstrated how making hardware emulation and native hardware indistinguishable is fundamentally infeasible. The emulator-based implementation of *DroidDungeon* inherits all limitations of the hardware emulation, but it can also be distributed on an actual device.

Fourth, we collected the events by stimulating each app with ARES. Thus, we inherit the limitations of the dynamic analysis [23, 73], namely, there are parts of the code that may not be explored.

Finally, as mentioned in Section 6.3, our measures in comparing execution traces should be considered a lower bound because there are evasive controls that our sandbox does not need to mitigate. In those cases, if there are differences, they are not measured.

Conclusions. This paper focuses on seeking, collecting, and measuring Android evasive techniques based on their behaviors and methods, both in malicious and benign apps. For this purpose, we developed *DroidDungeon*, a probe-based sandbox that jointly fulfills the anti-evasion, maintainability, and scalability criteria.

Our experiments show the primary purposes of evasive checks in malware and goodware, and our main result highlights that 14% of malware and 4% of goodware refrain from running if their evasive controls detect a potential analysis environment. It is crucial to consider these percentages when dealing with dynamic analysis of Android apps and, thus, consider the bias introduced by evasive controls, which this work sought to shed light on.

ACKNOWLEDGMENTS

We would like to express our sincere gratitude to Slasti Mormanti for his invaluable support during the development of this paper.

This work has benefited from a government grant managed by the National Research Agency under France 2030 with the reference "ANR-22-PECY-0007".

REFERENCES

[1] 2016. Android Hostile Environment Detection. https://github.com/Fuzion24/AndroidHostileEnvironmentDetection. Accessed December 2, 2023.

[2] 2018. Android Anti Debug. https://github.com/GToad/Android_Anti_Debug. Accessed December 2, 2023.

[3] 2020. 3 ways to detect the SELinux status in Android natively. https://erev0s.com/blog/3-ways-detect-selinux-status-android-natively/. Accessed December 2, 2023.

[4] 2021. Anti Debug and Memory Dump. https://github.com/darvincisec/AntiDebugandMemoryDump. Accessed December 2, 2023.

[5] 2021. GriftHorse Android Trojan Steals Millions from Over 10 Million Victims Globally. https://www.zimperium.com/blog/grifthorse-android-trojan-steals-millions-from-over-10-million-victims-globally/. Accessed December 2, 2023.

[6] 2022. Cybercriminals attack users with 400,000 new malicious files daily. https://www.kaspersky.com/about/press-releases/2022_cybercriminals-attack-users-with-400000-new-malicious-files-daily---that-is-5-more-than-in-2021. Accessed December 2, 2023.

[7] 2022. frida-detection. https://github.com/muellerberndt/frida-detection. Accessed December 2, 2023.

[8] 2022. Google Play Scraper. https://github.com/facundoolano/google-play-scraper. Accessed December 2, 2023.

[9] 2023. Al-Khaser. https://github.com/LordNoteworthy/al-khaser. Accessed December 2, 2023.

[10] 2023. Android Al-Khaser. https://gitlab.eurecom.fr/saonzo/AAl-Khaser. Accessed December 2, 2023.

[11] 2023. Android Verified Boot 2.0. https://android.googlesource.com/platform/external/avb/+/master/README.md. Accessed December 2, 2023.

[12] 2023. apkeep. https://github.com/EFForg/apkeep. Accessed December 2, 2023.

[13] 2023. AppLovin MAX. https://www.applovin.com/. Accessed December 2, 2023.

[14] 2023. Chartboost. https://support.chartboost.com/en. Accessed December 2, 2023.

[15] 2023. Flurry. https://www.flurry.com/. Accessed December 2, 2023.

[16] 2023. genuine. https://github.com/brevent/genuine. Accessed December 2, 2023.

[17] 2023. Ghidra. https://ghidra.re/. Accessed December 2, 2023.

[18] 2023. InMobi. https://www.inmobi.com/sdk. Accessed December 2, 2023.

[19] 2023. Magisk. https://github.com/topjohnwu/Magisk. Accessed December 2, 2023.

[20] 2023. mbc-markdown. https://github.com/MBCProject/mbc-markdown. Accessed December 2, 2023.

[21] 2023. readelf. https://man7.org/linux/man-pages/man1/readelf.1.html. Accessed December 2, 2023.

[22] 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: 21st International Conference, ISC 2018, Guildford, UK, September 9–12, 2018, Proceedings 21*. Springer, 47–66.

[23] Ashish Aggarwal and Pankaj Jalote. 2006. Integrating static and dynamic analysis for detecting vulnerabilities. In *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, Vol. 1. IEEE, 343–350.

[24] Aisha Ali-Gombe, Sneha Sudhakaran, Andrew Case, Golden G Richard III, Sencun Zhu, Peiyi Han, Thenkurussi Kesavadas, Dawu Gu, Kehuan Zhang, XiaoFeng Wang, et al. 2019. DroidScraper: a tool for Android in-memory object recovery and reconstruction. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*. 547–559.

[25] Jeremy Andrus, Christoffer Dall, Alexander Van't Hof, Oren Laadan, and Jason Nieh. 2011. Cells: a virtual mobile smartphone architecture. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 173–187.

[26] Simone Aonzo, Yufei Han, Alessandro Mantovani, and Davide Balzarotti. 2023. Humans vs. machines in malware classification. *Proc. of USENIX-23* (2023).

[27] Luciano Bello and Marco Pistoia. 2018. Ares: triggering payload of evasive android malware. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. 2–12.

[28] Harel Berger, Chen Hajaj, and Amit Dvir. 2020. Evasion is not enough: A case study of android malware. In *International Symposium on Cyber Security Cryptography and Machine Learning*. Springer, 167–174.

[29] Stefano Berlato and Mariano Ceccato. 2020. A large-scale study on the adoption of anti-debugging and anti-tampering protections in android apps. *Journal of Information Security and Applications* 52 (2020), 102463.

[30] Lorenzo Bordoni, Mauro Conti, and Riccardo Spolaor. 2017. Mirage: Toward a stealthier and modular malware analysis sandbox for android. In *Computer Security–ESORICS 2017: 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I 22*. Springer, 278–296.

[31] Software Freedom Conservancy. 2023. QEMU. https://www.qemu.org/. Accessed December 2, 2023.

[32] The MITRE Corporation. 2023. EvadeMe. https://attack.mitre.org/techniques/T1633/001/. Accessed December 2, 2023.

[33] Cryptomathic. 2022. Virtualization/Sandbox Evasion: System Checks. https://www.cryptomathic.com/news-events/blog/app-hardening-for-mobile-banking-and-payment-apps-emulator-detection. Accessed December 2, 2023.

[34] Yuning Cui, Yi Sun, and Zhaowen Lin. 2023. DroidHook: a novel API-hook based Android malware dynamic analysis sandbox. *Automated Software Engineering* 30, 1 (2023), 10.

[35] Deshun Dai, Ruixuan Li, Junwei Tang, Ali Davanian, and Heng Yin. 2020. Parallel space traveling: A security analysis of app-level virtualization in android. In *Proceedings of the 25th ACM Symposium on Access Control Models and Technologies*. 25–32.

[36] Savino Dambra, Yufei Han, Simone Aonzo, Platon Kotzias, Antonino Vitale, Juan Caballero, Davide Balzarotti, and Leyla Bilge. 2023. Decoding the Secrets of Machine Learning in Malware Classification: A Deep Dive into Datasets, Feature Extraction, and Model Performance. *arXiv preprint arXiv:2307.14657* (2023).

[37] Daniele Cono D'Elia, Emilio Coppa, Federico Palmaro, and Lorenzo Cavallaro. 2020. On the Dissection of Evasive Malware. *IEEE Transactions on Information Forensics and Security* 15 (2020), 2750–2765.

[38] Wael F Elsersy, Ali Feizollah, and Nor Badrul Anuar. 2022. The rise of obfuscated Android malware and impacts on detection methods. *PeerJ Computer Science* 8 (2022), e907.

[39] evilthreads669966. 2021. EvadeMe. https://github.com/evilthreads669966/evademe. Accessed December 2, 2023.

[40] Farnood Faghihi, Mohammad Zulkernine, and Steven Ding. 2022. CamoDroid: An Android application analysis environment resilient against sandbox evasion. *Journal of Systems Architecture* 125 (2022), 102452.

[41] Parvez Faruki, Rati Bhan, Vinesh Jain, Sajal Bhatia, Nour El Madhoun, and Rajendra Pamula. 2023. A Survey and Evaluation of Android-Based Malware Evasion Techniques and Detection Frameworks. *Information* 14, 7 (2023), 374.

[42] Stichting Cuckoo Foundation. 2023. Cuckoo Sandbox. https://cuckoosandbox.org/. Accessed December 2, 2023.

[43] Jyoti Gajrani, Jitendra Sarswat, Meenakshi Tripathi, Vijay Laxmi, Manoj Singh Gaur, and Mauro Conti. 2015. A robust dynamic analysis system preventing SandBox detection by Android malware. In *Proceedings of the 8th International Conference on Security of Information and Networks*. 290–295.

[44] Nicola Galloro, Mario Polino, Michele Carminati, Andrea Continella, and Stefano Zanero. 2022. A Systematical and longitudinal study of evasive behaviors in windows malware. *Computers & Security* 113 (2022), 102550.

[45] Tal Garfinkel, Keith Adams, Andrew Warfield, Jason Franklin, et al. 2007. Compatibility Is Not Transparency: VMM Detection Myths and Realities.. In *HotOS*.

[46] Genymobile. 2023. Genymotion. https://www.genymotion.com/. Accessed December 2, 2023.

[47] Google. 2018. Protecting WebView with Safe Browsing. https://android-developers.googleblog.com/2018/04/protecting-webview-with-safe-browsing.html. Accessed December 2, 2023.

[48] Google. 2023. Configuring ART. https://source.android.com/docs/core/runtime/configure. Accessed December 2, 2023.

[49] Google. 2023. Google Safe Browsing Service. https://developer.android.com/develop/ui/views/layout/webapps/managing-webview#safe-browsing. Accessed December 2, 2023.

[50] Google. 2023. Play Integrity API. https://developer.android.com/google/play/integrity. Accessed December 2, 2023.

[51] Google. 2023. Protect against security threats with SafetyNet. https://developer.android.com/training/safetynet. Accessed December 2, 2023.

[52] Skanda Hazarika. 2022. Xposed. https://www.xda-developers.com/best-xposed-modules/. Accessed December 2, 2023.

[53] Muhammad Ibrahim, Abdullah Imran, and Antonio Bianchi. 2021. Safetynot: on the usage of the safetynet attestation API in android. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*. 150–162.

[54] Irdeto. 2023. Denuvo Mobile Games Protection. https://irdeto.com/denuvo/mobile-games-protection/. Accessed December 2, 2023.

[55] Sainadh Jamalpur, Yamini Sai Navya, Perla Raja, Gampala Tagore, and G Rama Koteswara Rao. 2018. Dynamic malware analysis using cuckoo sandbox. In *2018 Second international conference on inventive communication and computational technologies (ICICCT)*. IEEE, 1056–1060.

[56] Yiming Jing, Ziming Zhao, Gail-Joon Ahn, and Hongxin Hu. 2014. Morpheus: automatically generating heuristics to detect android emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference*. 216–225.

[57] Michael Kerrisk. 2021. xdr. https://man7.org/linux/man-pages/man3/xdr.3.html. Accessed December 2, 2023.

[58] Brian Kondracki, Babak Amin Azad, Najmeh Miramirkhani, and Nick Nikiforakis. 2022. The droid is in the details: Environment-aware evasion of android sandboxes. In *Proceedings of the 29th Network and Distributed System Security Symposium (NDSS)*.

[59] Lang Liu, Yacong Gu, Qi Li, and Purui Su. 2017. RealDroid: Large-Scale Evasive Malware Detection on" Real Devices". In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 1–8.

[60] Check Point Software Technologies LTD. 2017. CuckooDroid. https://github.com/idanr1986/cuckoo-droid. Accessed December 2, 2023.

[61] Jining Luohe Network Technology Co. Ltd. 2020. VirtualApp. https://github.com/asLody/VirtualApp Accessed online: December 2, 2023.

[62] Tongbo Luo, Cong Zheng, Zhi Xu, and Xin Ouyang. 2017. Anti-plugin: Don't let your app play as an android plugin. *Proceedings of Blackhat Asia* (2017).

[63] Lorenzo Maffia, Dario Nisi, Platon Kotzias, Giovanni Lagorio, Simone Aonzo, and Davide Balzarotti. 2021. Longitudinal Study of the Prevalence of Malware Evasive Techniques. *arXiv preprint arXiv:2112.11289* (2021).

[64] Dominik Maier, Tilo Müller, and Mykola Protsenko. 2014. Divide-and-conquer: Why android malware cannot be stopped. In *2014 Ninth International Conference on Availability, Reliability and Security*. IEEE, 30–39.

[65] Guozhu Meng, Yinxing Xue, Chandramohan Mahinthan, Annamalai Narayanan, Yang Liu, Jie Zhang, and Tieming Chen. 2016. Mystique: Evolving android malware for auditing anti-malware tools. In *Proceedings of the 11th ACM on Asia conference on computer and communications security*. 365–376.

[66] Alessio Merlo, Antonio Ruggia, Luigi Sciolla, and Luca Verderame. 2021. You shall not repackage! demystifying anti-repackaging on android. *Computers & Security* 103 (2021), 102181.

[67] Najmeh Miramirkhani, Mahathi Priya Appini, Nick Nikiforakis, and Michalis Polychronakis. [n. d.]. Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts. In *2017 IEEE Symposium on Security and Privacy (SP)*. 1009–1024.

[68] Samrah Mirza, Haider Abbas, Waleed Bin Shahid, Narmeen Shafqat, Mariagrazia Fugini, Zafar Iqbal, and Zia Muhammad. 2021. A malware evasion technique for auditing android anti-malware solutions. In *2021 IEEE 30th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. IEEE, 125–130.

[69] Dario Nisi, Antonio Bianchi, and Yanick Fratantonio. 2019. Exploring {Syscall-Based} Semantics Reconstruction of Android Applications. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. 517–531.

[70] NowSecure. 2023. Frida. https://frida.re/. Accessed December 2, 2023.

[71] 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 USENIX Workshop on Offensive Technologies (WOOT)*, Vol. 41. 86.

[72] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. 2014. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the seventh european workshop on system security*. 1–6.

[73] Andrey Petukhov and Dmitry Kozlov. 2008. Detecting security vulnerabilities in web applications using dynamic analysis with penetration testing. *Computing Systems Lab, Department of Computer Science, Moscow State University* (2008), 1–120.

[74] pjlantz. 2019. DroidBox. https://github.com/pjlantz/droidbox. Accessed December 2, 2023.

[75] Zhengyang Qu, Shahid Alam, Yan Chen, Xiaoyong Zhou, Wangjun Hong, and Ryan Riley. 2017. DyDroid: Measuring dynamic code loading and its security implications in Android applications. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 415–426.

[76] rednaga. 2023. APKiD. https://github.com/rednaga/APKiD. Accessed December 2, 2023.

[77] Andrea Romdhana, Alessio Merlo, Mariano Ceccato, and Paolo Tonella. 2022. Deep reinforcement learning for black-box testing of android apps. *ACM Transactions on Software Engineering and Methodology* (2022).

[78] Antonio Ruggia, Andrea Possemato, Alessio Merlo, Dario Nisi, and Simone Aonzo. 2023. Android, Notify Me When It Is Time To Go Phishing. In *EUROS&P 2023, 8th IEEE European Symposium on Security and Privacy*.

[79] Onur Sahin, Ayse K Coskun, and Manuel Egele. 2018. Proteus: Detecting android emulators from instruction-level profiles. In *Research in Attacks, Intrusions, and Defenses: 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings 21*. Springer, 3–24.

[80] samohyes. 2018. Anti-vm-in-Android. https://github.com/samohyes/Anti-vm-in-android. Accessed December 2, 2023.

[81] scottyab. 2021. RootBeer. https://github.com/scottyab/rootbeer. Accessed December 2, 2023.

[82] Silvia Sebastián and Juan Caballero. 2020. Avclass2: Massive malware tag extraction from av labels. In *Annual Computer Security Applications Conference*. 42–53.

[83] OWASP Mobile Application Security. 2023. Android Anti-Reversing Defenses. https://mas.owasp.org/MASTG/Android/0x05j-Testing-Resiliency-Against-Reverse-Engineering/. Accessed December 2, 2023.

[84] Luman Shi, Jianming Fu, Zhengwei Guo, and Jiang Ming. 2019. " Jekyll and Hyde" is Risky: Shared-Everything Threat Mitigation in Dual-Instance Apps. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*. 222–235.

[85] IBM Mobile Enterprise Software. 2018. evadroid. https://bitbucket.org/IBMmobile/evadroid/src/master/. Accessed December 2, 2023.

[86] Wenna Song, Jiang Ming, Lin Jiang, Yi Xiang, Xuanchen Pan, Jianming Fu, and Guojun Peng. 2021. Towards transparent and stealthy android os sandboxing via customizable container-based virtualization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2858–2874.

[87] San-Tsai Sun, Andrea Cuadros, and Konstantin Beznosov. 2015. Android rooting: Methods, detection, and evasion. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*. 3–14.

[88] Kimberly Tam, Aristide Fattori, Salahuddin Khan, and Lorenzo Cavallaro. 2015. Copperdroid: Automatic reconstruction of android malware behaviors. In *NDSS Symposium 2015*. 1–15.

[89] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. 2017. The evolution of android malware and android analysis techniques. *ACM Computing Surveys (CSUR)* 49, 4 (2017), 1–41.

[90] DroidPlugin Team. 2020. DroidPlugin. https://github.com/DroidPluginTeam/DroidPlugin Accessed online: December 2, 2023.

[91] 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*. 447–458.

[92] VirusTotal. 2023. VirusTotal. https://www.virustotal.com. Accessed December 2, 2023.

[93] Jue Wang, Yepang Liu, Chang Xu, Xiaoxing Ma, and Jian Lu. 2016. E-greenDroid: effective energy inefficiency analysis for android applications. In *proceedings of the 8th Asia-Pacific Symposium on Internetware*. 71–80.

[94] Yifang Wu, Jianjun Huang, Bin Liang, and Wenchang Shi. 2020. Do not jail my app: Detecting the Android plugin environments by time lag contradiction. *Journal of Computer Security* 28, 2 (2020), 269–293.

[95] Lei Xu, Guoxi Li, Chuan Li, Weijie Sun, Wenzhi Chen, and Zonghui Wang. 2015. Condroid: a container-based virtualization solution adapted for android devices. In *2015 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*. IEEE, 81–88.

[96] Lok-Kwong Yan and Heng Yin. 2012. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis.. In *USENIX security symposium*. 569–584.

[97] Lei Zhang, Zhemin Yang, Yuyu He, Mingqi Li, Sen Yang, Min Yang, Yuan Zhang, and Zhiyun Qian. 2019. App in the middle: Demystify application virtualization in Android and its security threats. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 3, 1 (2019), 1–24.

[98] Cong Zheng, Tongbo Luo, Zhi Xu, Wenjun Hu, and Xin Ouyang. 2018. Android plugin becomes a catastrophe to Android ecosystem. In *Proceedings of the First Workshop on Radical and Experiential Security*. 61–64.

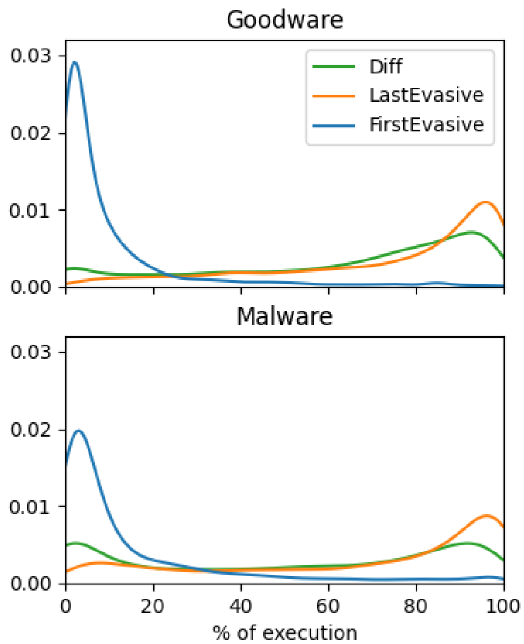

Figure 2: Kernel Density Estimation for the timing of DET and IET controls in malware and goodware.

A APPENDIX

Table 2: Category to system calls

Syscall category	System call
ACCESS_FILE	(f)access(at), open(at), *stat*, readlink(at), getdents64
EXE_COMMAND	execve(at)
PROCESS_MNG	clone, (v)fork, kill, wait(id|4), ptrace, pipe(2), tee, mq_*, sched_*, rt_sigaction, signalfd(64)
FS_MNG	getcwd, (f)chdir, renameat(2), mkdir(at), rmdir, *link*, *chmod*, mknod(at), inotify_add_watch, fanotify_mark, (p)poll, *xattr, flock, mount
NETWORK	socket, binde, accept(4), connect, getsockname, listen, getpeername, sendto, recvfrom, sendmsg, socketpair, setsockopt
SYSTEM	reboot, getcpu, sys*, uname, time*, clock_*
IDENTITY	*gid, *pid, *uid, *sid
MEMORY_MNG	mmap, *mprotect

Table 3: Mapping between SafetyNet Binder and high-level security mechanisms.

Id	High-level check	Retrived data
7	Attestation API	`AttestationResponse`
4 14	Verify Apps API	`VerifyAppsUserResponse`
5 13		`HarmfulAppsResponse`
12 3	Safe Browsing API	`InitSafeBrowsingResponse` `SafeBrowsingResponse`
6	reCAPTCHA API	`RecaptchaTokenResponse`

Listing 1: Android kernel modification to enable system call tracepoint in an external kernel module.

```
// register_trace_sys_enter
EXPORT_TRACEPOINT_SYMBOL_GPL(sys_enter);
// register_trace_sys_exit
EXPORT_TRACEPOINT_SYMBOL_GPL(sys_exit);
```

Table 4: Best correlation between evasive techniques and packers.

Packer	% packed apps	# evasive	Most used evasive	
APKProtect	5.2%	43	EMU-SYSTEM-PROPS	92%
ApkEncryptor	4.0%	55	SIGNATURE-ZIP-FILE	87%
Baidu	5.8%	37	HOOK-PROC_ART-MAPS	87%
Bangcle	3.3%	9	HOOK-FRIDA-FILE	65%
Bangcle (SecShell)	1.6%	12	HOOK-FRIDA-FILE	84%
DexProtector	4.1%	51	HOOK-STACKTRACE-API	88%
DexProtector for AIDE	1.2%	48	NET-SSL_PINNING-API	88%
Ijiami	4.3%	27	SIGNATURE-ZIP-FILE	90%
Jiagu	43.0%	56	HOOK-FRIDA-FILE	69%
Mobile Tencent Protect	9.9%	43	HOOK-FRIDA-FILE	88%
MultidexPacker	5.3%	26	SIGNATURE-ZIP-FILE	100%
SecNeo.A	1.9%	24	HOOK-PROC_ART-MAPS	58%
Tencent's Legu	2.3%	29	SIGNATURE-ZIP-FILE	93%
Unicom SDK Loader	5.1%	46	HOOK-STACKTRACE-API	50%

Table 5: Stats about trace comparison in evasive samples, all numbers are in percentage %

	Malware					Goodware				
	$R = B$	$R \subset B$	$\|B \setminus R\| > \|R \setminus B\|$	$\|B\| > 0 \wedge \|R\| = 0$	*avg* $\|R \setminus B\| \div \|B \setminus R\|$	$R = B$	$R \subset B$	$\|B \setminus R\| > \|R \setminus B\|$	$\|B\| > 0 \wedge \|R\| = 0$	*avg* $\|R \setminus B\| \div \|B \setminus R\|$
a11y	68.01%	10.70%	1.62%	0.00%	62.98	89.27%	4.43%	0.42%	0.00%	61.08
BR	85.90%	9.37%	0.65%	0.14%	93.84	92.38%	4.31%	0.00%	2.94%	0.00
CLI	90.78%	3.65%	1.82%	0.30%	97.22	98.19%	0.00%	1.20%	0.00%	100.0
CP	79.79%	13.28%	1.08%	0.03%	72.14	88.68%	6.23%	0.21%	0.21%	90.00
DAPI	73.60%	9.97%	6.96%	0.37%	83.01	95.56%	1.50%	0.06%	0.17%	100.0
DCL	86.92%	1.88%	0.21%	0.05%	87.50	80.22%	0.86%	0.00%	0.24%	0.00
FS	23.21%	10.47%	34.78%	0.00%	61.43	26.63%	5.69%	28.19%	0.00%	73.46
IPC	74.58%	10.77%	2.96%	0.25%	83.11	78.87%	9.73%	0.56%	0.45%	91.70
NET	48.50%	6.03%	31.48%	0.46%	87.45	29.74%	2.85%	52.45%	0.86%	93.93
PERM	80.37%	10.27%	1.68%	0.00%	83.72	80.01%	4.44%	1.92%	0.00%	90.74
SS	67.59%	25.35%	1.32%	0.03%	84.72	80.15%	4.54%	0.18%	0.00%	100.0

Table 6: Comparison with SOTA w.r.t. the anti-evasion, maintainability, and scalability criteria.

Requirement	Container-Based			Emulator-Based			
	Cells [25]	Condroid [95]	VPBox [86]	VMI-based [88, 96]	Framework mod. [59, 74]	Hook-based [30, 34, 40, 43, 60]	*DroidDungeon*
Anti-evasion	◐	◐	●	○	●	◐	●
Maintenability	○	○	○	◐	○	◐	●
Scalability	○	○	○	◐	●	●	●