Android's Cat-and-Mouse Game: Understanding Evasion Techniques against Dynamic Analysis

Shuang Li*†, Rui Li*†, Shishuai Yang*†, and Wenrui Diao*†(✉)
*School of Cyber Science and Technology, Shandong University
{lishuang128, leiry, shishuai}@mail.sdu.edu.cn, diaowenrui@link.cuhk.edu.hk
†Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University

***Abstract*—The Android OS, known for its openness and flexibility, dominates the global smartphone market, enabling the creation and distribution of a vast array of apps. However, this openness also attracts malicious apps that threaten user security. To counter these threats, static and dynamic analysis techniques are employed. Despite these efforts, evasion techniques such as code obfuscation and anti-debugging are increasingly used to bypass these analyses.**

In this study, we conduct a comprehensive review of current evasion and anti-evasion techniques and assess their real-world impact by analyzing 108,099 benign apps, 11,730 malicious apps, and 11 online dynamic analysis platforms. Our findings reveal that 68.1% of apps employ evasion techniques, with benign apps using them more frequently than malicious ones. Malicious apps, however, demonstrate more cautious behaviors when evading dynamic analysis. Additionally, our evaluation of dynamic analysis platforms shows that most evasion techniques, including simple methods like checking fields in the `Build` class, successfully evade detection, indicating a significant gap in current anti-evasion capabilities. Our research provides critical insights into the ongoing battle between Android app security and evasion techniques, underscoring the need for improved countermeasures to enhance user security.

I. Introduction

With the rapid progression of mobile technology, smartphones have become a fundamental component of our daily lives. The Android OS, known for its openness and flexibility, has achieved a dominant position in the global smartphone market. It provides a comprehensive suite of APIs, enabling developers to create various apps that meet various market needs. These apps are then distributed through Google Play [12] or alternative third-party marketplaces, providing users worldwide with easy access.

However, with the continuous development of the Android system, malicious apps have also been increasing in its ecosystem. These malicious apps not only degrade the user experience but, more critically, threaten the security of user data. To combat these threats, automated static and dynamic analysis techniques are extensively employed by security analysts [27], [37] or analysis platforms [21], [6]. Static analysis [40] involves scrutinizing the app's code without executing it, utilizing decompilation and disassembly for security assessments. Conversely, dynamic analysis [25] involves observing the app's behavior during execution, utilizing techniques such as execution tracing with debuggers or monitoring in a controlled sandbox environment to detect suspicious behaviors.

Evading Analysis. In response, malicious apps have developed strategies to evade these analyses. To evade static analysis, obfuscation and packer techniques are frequently adopted as well-recognized standard measures [34]. Conversely, a wider variety of techniques are employed to evade dynamic analysis. Previous research [49], [44], [41], [38], [33], [31], [50], [46], [39] has extensively documented evasion techniques, highlighting that automated analysis systems are usually built on emulators (virtual machines) for better isolation and behavior monitoring. Similarly, benign app developers can use these evasion techniques to protect intellectual property rights and prevent unauthorized copying. This leads to a continuous struggle between analysis tools and apps.

In contrast, relatively little work has focused on detecting these evasion techniques. Afonso et al. [26] identified evasion techniques by comparing the execution traces of apps on physical versus virtual devices, successfully detecting a portion of the evasion samples. However, this study examined only a small number of malicious apps, and the limitations of dynamic analysis could lead to missed identifications. Berlato et al. [28] conducted a large-scale study of Android apps but focused solely on anti-debugging and anti-tampering protection techniques. Consequently, there is currently a lack of a comprehensive large-scale evaluation of the use of evasion techniques against dynamic analysis in Android.

Furthermore, as part of the ongoing cat-and-mouse game, research has explored countermeasures against app evasion, such as BareDroid, developed by Simone et al. [42]. BareDroid is a bare-metal analysis framework for Android apps that effectively counters evasion techniques. Dynamic analysis platforms can employ these countermeasures to perform effective analyses on malicious apps. However, it remains uncertain whether online dynamic analysis platforms have implemented the anti-evasion techniques proposed in these countermeasures and whether they can address app evasion issues as effectively in practice.

Our Work. To address the research gaps and fully understand the current status of the dynamic evasion techniques competition in Android, we conducted a comprehensive literature review, summarizing all current dynamic analysis evasion techniques and anti-evasion measures for automated analysis platforms. Building on this knowledge, we further examined the real-world usage of dynamic analysis evasion techniques in a large-scale study and evaluated the effectiveness of current automated dynamic analysis platforms' anti-evasion technologies. Our large-scale measurement, covering 108,099 benign apps, 11,730 malicious apps, and 11 online automated dynamic analysis platforms, focuses on answering the following three questions:

RQ1 ***How do evasion techniques against dynamic analysis deploy in the wild?***
Most apps (68.1%) distributed through the Google Play Market employ evasion techniques. App developers typically implement these techniques directly within the app's local Java / Kotlin code, prioritizing methods that are simple to deploy and have minimal impact on app performance.

RQ2 ***What are the differences between evasion techniques deployed by benign and malicious apps?***
Only 15.8% of malicious apps use techniques to evade dynamic analysis, a proportion significantly lower than that of benign apps. The evasion behaviors of malicious apps are often more cautious, with a common reaction being to directly terminate the app.

RQ3 ***Can existing evasion techniques effectively evade the detection of online automated dynamic analysis platforms?***
Most evasion techniques, including some straightforward methods such as checking fields in the `Build` class, can successfully identify and evade these dynamic platforms. This reflects the lack of effective anti-evasion measures in current online dynamic analysis platforms.

Contributions. The main contributions of this paper include:

- **Comprehensive Review.** We conducted a comprehensive review and summarized all current techniques for evading dynamic analysis, as well as the corresponding measures for countering evasion in automated analysis platforms. This fills a critical knowledge gap in the field.
- **Large-Scale Empirical Study.** We performed a large-scale empirical analysis involving 108,099 benign apps, 11,730 malicious apps, and 11 online automated dynamic analysis platforms. This provides a panoramic view of the deployment and effectiveness of evasion techniques in the real world.
- **Systematic Evaluations.** We systematically evaluated the differences in evasion techniques used by benign and malicious apps, as well as the capabilities of current online dynamic analysis platforms to counter these techniques. This highlights the need for improved anti-evasion measures in existing analysis platforms.

Data Availability. The raw measurement data is available at https://doi.org/10.5281/zenodo.11192208.

II. Background

In this section, we provide the necessary background knowledge on automated analysis techniques and summarize the techniques proposed in existing studies for Android apps to evade dynamic analysis.

A. Automated App Analysis

Automated analysis techniques can be divided into static and dynamic analysis techniques. These techniques play a crucial role in software security, offering significant advantages over traditional manual testing methods. They not only reduce the workload of security analysts but also improve the efficiency and accuracy of malware detection.

Static Analysis. Static analysis technique can be used to scrutinize the structure, content, and composition of an app's source code, bytecode, binaries, or even decompiled code without running the program [40]. It includes various sophisticated methods such as decompilation and data flow analysis [3]. Through static analysis, researchers can uncover vulnerabilities, ensure compliance with coding standards, and assess the security posture of apps by analyzing the code for malware patterns or by evaluating the permissions requested by the app against common misuse patterns.

Dynamic Analysis. Dynamic analysis technique can be used to observe and evaluate the execution process of a program while it is running [25]. It includes monitoring the running behavior of apps, capturing logs, and using automated interaction tools that simulate user operations to trigger and evaluate specific functionalities of an app. Dynamic analysis is particularly effective in assessing how apps respond to various operational scenarios, such as their interactions with system components, network communications, and user inputs.

Evading Analysis. Android apps, aware of scrutiny from automated analysis, have developed various strategies to evade detection. In the context of static analysis, developers might use techniques such as code obfuscation, data encryption, and dynamic code loading. These methods aim to obscure the app's logic, making it difficult for static analysis tools to interpret the code and accurately identify potential security threats. Code obfuscation and data encryption transform the code into a form that is hard for humans and automated tools to understand, while dynamic code loading allows parts of the app to be loaded or modified during runtime, thereby evading static detection mechanisms [34].

In the context of dynamic analysis, the Android community has seen various evasion techniques emerge. These techniques are designed to detect and respond to the presence of automated analysis environments, thereby avoiding detection or altering the app's behaviors under scrutiny (e.g., not loading the malicious payloads). Such strategies include *environment detection*, where apps check for signs of being run in a simulator or a sandboxed environment, and *analysis obstruction*, where apps directly interfere with the normal operation of dynamic analysis tools. The subsequent subsections will detail these dynamic analysis evasion techniques.

B. Classifications of Evasion Techniques

To comprehensively cover existing evasion techniques against dynamic analysis, we systematically scrutinize relevant literature. This process involved two sub-processes: literature search and review. For the search process, we conducted a systematic keyword search via Google Scholar. The keywords are strategically selected to construct a comprehensive search query. Specifically, we empirically summarized and heuristically identified commonly used keywords by analyzing several relevant publications. Then, we manually verified and optimized the keyword list during the literature review process, ultimately determining the most suitable combination of keywords. The keywords we used can be categorized into three groups, which form our search queries in various combinations:

- G1 (Scenarios): *Android*, *mobile*, *app*, *malware*.
- G2 (Action): *evade*, *evasion*, *bypass*, *hide*, *escape*.
- G3 (Objective): *dynamic analysis*, *sandbox*, *runtime*, *emulat**, *VM*, *virtual*, *detection*.

For the review process, we carefully review the abstracts of the retrieved publications to determine their relevance to our research objectives. This preliminary filtering process ensured that we only considered literature closely related to our research focus. We listed these relevant publications in section VII. By reading the full texts of these papers, we conducted a comprehensive summary and classification of the techniques proposed for evading dynamic analysis. Additionally, we refined the search keyword list based on the insights gained during this process.

Emulator Detection (ED). Security analysts typically run apps in virtual machines, emulators, or sandboxes to conduct testing and research safely in a controlled environment. However, making simulated environments indistinguishable from real devices is extremely difficult, so differences between simulated environments and real devices still exist. Therefore, apps can detect certain information that differs between virtual environments and real devices, thereby identifying the virtual environment and attempting to evade it.

- *System Property (ED1).* Apps can use the values of these system properties to determine whether they are running in a virtual environment [38], [44], [41]. For instance, the property `ro.product.brand` identifies the device brand. If this property's value is `"generic"`, it indicates a virtual environment; otherwise, it suggests a real device.
- *Hardware Component (ED2).* Apps can identify simulated environments by examining hardware-related information such as CPU serial numbers, CPU frequency, sensors, and battery details [49], [44], [41], [38], [39].
- *Software Component (ED3).* Apps can determine if they are running in a simulated environment by detecting the presence of real device-specific apps and supporting software for Google Internet Services [49], [38].
- *Network (ED4).* Apps can detect simulated environments by evaluating the device's network address starting with `10.0.2/24` and checking whether a ping test is successful in testing the ICMP communications [49], [44], [41].
- *Traces of Usage (ED5).* Apps can determine if they are operating in a virtual environment by analyzing user usage trace data. Real devices often show signs of "wear and tear" due to user interactions, a feature typically absent in most Android sandboxes [41], [31], [39].
- *Specific Emulator Fingerprints (ED6).* Apps can check whether artifacts of common simulation environments exist on the device [44]. For example, `/sys/qemu_trace` will exist in the QEMU environment.
- *Performance (ED7).* Apps can test the device's CPU and graphics performance to determine whether they are currently in a simulation environment [49].

Anti-Debugging (AD). Debuggers can be used to analyze apps dynamically. To counteract this, app developers have implemented anti-debugging strategies, which can be categorized into two main approaches: anti-debugger mechanisms and detecting the app's debuggable state [30], [46], [48][1].

- *Debugger Detection (AD1).* There are two debugging protocols in Android, and two debuggers can be attached accordingly: one is the JDWP debugger, and the other is the GDB debugger. App developers can evade dynamic analysis by detecting both debuggers or preventing them from running. For the JDWP debugger, app developers can use the `Debug.isDebuggerConnected` API in Java code to detect its existence. Besides, they can modify variables related to the JDWP debugger in Native code to prevent it from running normally. Specifically, in Dalvik, apps can tamper with the pointer of the `DvmGlobals` structure in Native code by modifying the global variable `gDvm`. In ART, apps can override JDWP method pointers to achieve the same purpose. For the GDB debugger, the app can check whether there is an attached process by reading the `TracerPid` value in the `/proc/self/status` file, or it can attach a simulated debugger process to itself to prevent being debugged by the real debugger.
- *Debuggable Status Detection (AD2).* If an attacker wants to allow JDWP debugging, he must change the value of the debuggable flag in the app manifest file. App developers can check whether the value of this flag has been changed through the `ApplicationInfo.FLAG_DEBUGGABLE` or `BuildConfig.DEBUG` API. Moreover, if the device itself is debuggable, the app can be debugged regardless of its debuggable state. Apps can determine if the device is in a debuggable state by checking the value of the `ro.debuggable` system property.

Anti-Hooking (AH). Dynamic analysts can hook an app's API calls through the hook framework, such as Frida, to insert custom functionality while the app is running [50], [46]. For example, the hook technique allows developers to modify a method's parameters and return value or even completely replace the method's implementation. Currently, anti-hook methods in apps can be implemented by detecting the fingerprint of hook frameworks. At the Java code level, apps can check the stack trace to check Xposed by throwing an exception or traverse the list of running processes to check whether the Frida server is running. Besides, a hook framework may modify Java methods in an app to Native methods. Therefore, the app

[1]Given that some techniques of anti-debugging and anti-hooking are widely known but not documented in formal literature, we sought and identified relevant literature that mentions these techniques as references for our work.

can detect the presence of hook frameworks by checking if the attributes of all Java methods have changed using the `Modifier.isNative` method. At the Native code level, apps can ping the TCP port 27047 used by the Frida server by default to see whether it is open. In addition, apps can check whether hook framework-related libraries are mapped in memory.

Turing Test (TT). To enhance the coverage of the execution paths and thus more comprehensively test apps, some dynamic analysis frameworks have adopted automated exploration techniques, such as simulating UI interactions. However, there are significant differences in how human users and machine testers interact with apps, which allows the apps to detect automated analyses. We liken this detection mechanism based on interaction differences to a "Turing test," where an app attempts to distinguish whether its interlocutor is human or a machine.

- *Monkey Detection (TT1).* Monkey [16] is an automated testing tool provided by Google that can generate pseudo-random streams of user events. It can be identified by calling the `isUserAMonkey()` API and checking its return value.
- *Simulated Event Detection (TT2).* Some properties of `MotionEvent` and `KeyEvent` can reflect the gap between dynamic testing tools and humans. Additionally, dynamic analyzers are identified by measuring the frequency of event injection.
- *UI Trap Setting (TT3).* Apps can set UI traps that are inaccessible to human users but indistinguishable from dynamic analyzers, such as an unused but exported Activity or a valid but "invisible" control.

III. Methodology

In this section, we present our methodology designed to answer the research questions proposed in Section I. As depicted in Figure 1, our methodology can be structured into three distinct steps.

Step 1 **Dataset Construction.** Initially, we collected and summarized dynamic evasion techniques used in Android apps from current related publications and constructed corresponding fingerprints for these techniques. Following this, we gathered both benign and malicious APK datasets and preprocessed these apps to extract their key components for further analysis.

Step 2 **App Measurement Analysis.** Based on the collected datasets, we conducted an in-depth detection and analysis of the dynamic evasion techniques implemented in Android apps, aiming to understand their usage status and impact in the wild.

Step 3 **Test of Online Automated Dynamic Analysis Platforms.** Finally, we evaluated the ability of online automated dynamic analysis platforms to detect or defend the evasive behaviors in apps and obtained their overall strategies for combating dynamic analysis evasion behavior.

A. Dataset Construction

We need to construct two types of datasets for this study, as follows:

- *Evasion Technique Fingerprint Dataset.* As shown in Section II, we systematically summarized all known evasion techniques by conducting a comprehensive review of the current literature on Android apps' evasion against dynamic analysis. Based on this, we constructed a corresponding detection fingerprint for each evasion technique. We observed that evasion techniques utilizing environment detection are typically implemented through conditional statements. Therefore, we constructed a fingerprint tuple for each technique, with two elements representing the values on either side of the conditional statement. We call the first element the ***analysis fingerprint***, which signals us to conduct further analysis when this element appears. We call the second element the ***confirmation fingerprint***, which, if used in a conditional statement with the first element, indicates the presence of an evasion technique. For example, if an app detects whether it is running in a virtual environment by checking if the `BRAND` field value of the `Build` class is `"generic"` (ED1), the fingerprint of this evasion technique is defined as `(Build.BRAND (analysis fingerprint), "generic" (confirmation fingerprint))`. For techniques that evade dynamic analysis by obstructing the analysis process, like AD1, their mere presence indicates evasion. Therefore, we only construct an analysis fingerprint (which also serves as an identification fingerprint) for them[2].
- *APK Dataset.* To address research questions RQ1 and RQ2, we need to access a large-scale APK dataset. RQ1 focuses on evaluating the current status of evasion techniques in the wild, which requires us to observe across a broad and authentic range of app scenarios. Within the Android system, Google Play is not only the largest app distribution channel but also covers the vast majority of Android users globally and is the official app marketplace. By analyzing apps on Google Play, we can obtain observations that are close to real-world environments. For RQ2, our goal is to explore the differences between benign and malicious apps in evading dynamic analysis. The strict review process of Google Play means that its apps are generally benign, which facilitates our analysis of benign apps. Therefore, we requested access to the AndroZoo [2] dataset, an extensive archive of Android apps that has collected millions of APKs from Google Play and other sources. Ultimately, we successfully obtained a total of 108,099 apps from Google Play and 11,730 malicious apps. Every app in the malicious dataset has been identified as "malicious" by at least 35 antivirus engines. Subsequently, we extracted `dex` and `so` files from these APKs and used dex2jar [4] to convert `dex` files into `jar` files for further analysis.

[2]Due to the diverse and complex implementation of ED7 and TT3, we cannot generate their standardized detection fingerprints. Therefore, our fingerprint dataset does not include these two types.

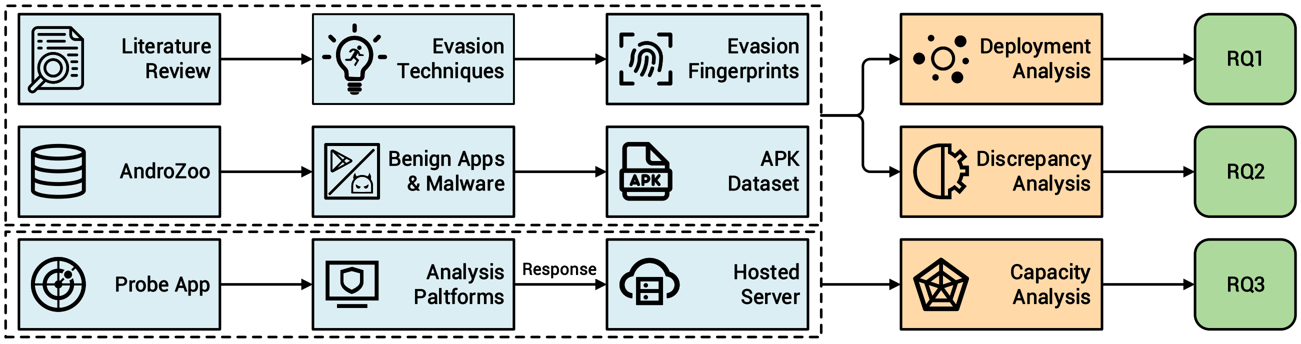

Fig. 1: Overview measurement flow.

B. App Measurement Analysis

Based on the deployment location of the evasion techniques, we divided the analysis of APKs into two layers: the Java / Kotlin layer and the Native layer. At the Java / Kotlin layer, we used the WALA [23] framework to analyze the extracted `jar` files. Specifically, we performed data flow analysis to locate and trace evasion techniques. At the Native layer, we built Yara [7] rules based on the fingerprints of evasion techniques to scan the extracted `so` files. These rules enable us to match the corresponding evasion fingerprints.

Data Flow Analysis. Since evasion techniques based on detection are typically implemented through conditional statements, the core of our data flow analysis is to locate the conditional statements associated with these techniques accurately. This process can be divided into the following three steps.

(1) *Initial taint location.* Firstly, to activate the whole analysis, our analysis framework locates the initial analysis points (taints) based on the analysis fingerprints of evasion techniques. Depending on the types of evasion fingerprints, our analysis framework categorizes initial taints into three types: field, method, and constant.

- *Field.* An app implements an evasion technique by obtaining a field value and making judgments based on it. For example, an app checks whether the value of the `DEVICE` field in the `Build` class is `"generic"` to determine if it is running in a virtual environment. For this type, our analysis framework traverses all statements in apps that obtain field values. If a field is the analysis fingerprint in our evasion fingerprint tuples, we will mark the variable to which the field value is assigned as the initial taint.
- *Method.* An app implements an evasion technique through the values returned by method calls. For this type, our analysis framework traverses all invocation statements in apps. If the method called is the analysis fingerprint in our evasion fingerprint tuples, our analysis framework will mark the return variable of this call as the initial taint.
- *Constant.* An app processes constant values for judgment to implement an evasion technique. For example, an app might retrieve system properties via reflection to make a judgment, and the system property value is constant. For this type, our analysis framework traverses all constants used in the app. If a constant matches the analysis fingerprint, our analysis framework will identify the variable where this constant was assigned and consider it as the initial taint.

(2) *Taint tracking.* Secondly, to confirm whether the taint involves evasion techniques, our analysis framework tracks it to a conditional statement and performs fingerprint recognition of the corresponding evasion technique. Our analysis framework employs a two-tiered taint tracking strategy: intra- and inter-procedural analysis, supplemented with Class Hierarchy Analysis (CHA) to construct the app's call graph.

Intra-procedural taint tracking. During this phase, our analysis framework conducts a detailed analysis of each method containing an initial taint. The initial taint might be located through the *initial taint location* or propagated through inter-procedural analysis. Our analysis framework uses a Control Flow Graph (CFG) to trace the propagation of the taint among statements within the analyzed method until it reaches a conditional statement. Then, our analysis framework evaluates the value on the other side of the condition to determine if it matches the confirmation fingerprint, thus assessing if an evasion technique is implemented.

Inter-procedural Taint Tracking. When a taint is passed to a field, a return statement, or method parameters, it triggers inter-procedural taint tracking. If a taint propagates to a field, our analysis framework will search for all methods that access this field and designate these methods as new analysis points, with the variable at the field assignment as the new initial taint. If a taint propagates to a return statement, the methods calling the current method will be treated as new analysis points. The return value at the point of the current method's call is set as the new initial taint. If a taint transfers to a method parameter, the called method will become a new analysis point, with the corresponding parameter as the new initial taint.

(3) *Third-party library determination.* Finally, considering that evasion techniques might also be deployed in third-party libraries, which may not necessarily be called, our analysis framework identifies whether they are deployed in third-party libraries and, if so, whether they are invoked by the APK's local code. This step can help us evaluate the actual usage status and identify the usage patterns of evasion techniques in apps. To achieve this, our analysis framework integrates

TABLE I: Statistics on Google Play apps using evasion techniques.

Evasion Techniques	ED1	ED2	ED3	ED4	ED5	ED6	AD1	AD2	AH	TT1	TT2	All*
All	66,407	37,988	43,989	5,007	72,574	16,800	31,876	47,047	1,746	8,407	16,250	87,038
TiE⋆	50/50	0/50	0/50	38/50	0/50	35/50	50/50	50/50	39/50	50/50	0/50	-
Java / Kotlin	62,219	-	-	1,940	-	12,823	29,947	46,873	1107	8,407	-	69,212
Native	8,459	-	-	2,016	-	3,198	2,567	670	201	-	-	12,610
All (adjusted)	66,407	-	-	3,950	-	15,745	31,876	47,047	1,283	8,407	-	73,607
Percentage	61.4%	-	-	3.7%	-	14.6%	29.5%	43.5%	1.2%	7.8%	-	68.1%

⋆: **T**echniques **i**ndeed **E**vading Dynamic Analysis in Practice
*: Some apps implement the evasion actions in both Java/Kotlin and Native code. Therefore, the [All] amount is not [Java/Kotlin] + [Native] directly.

LiteRadar [1], which is designed to inspect third-party libraries in APK files.

Note that, as stated in Section III-A, some techniques evade dynamic analysis by obstructing the analysis process, and we only constructed their analysis fingerprints. Therefore, when detecting such techniques, we can skip Step (2), which aims at matching the confirmation fingerprint.

Yara Rules Scan. We constructed the corresponding Yara [7] rules based on our fingerprints to scan the app's native code. Yara is a powerful open-source tool designed to help malware researchers identify and classify malware samples. Using Yara requires creating malware signature rules that contain text or binary patterns. As shown in Listing 1, these rules are formulated based on a combination of string sets and Boolean expressions (Line 5, 8), effectively describing the characteristics of malware. Specifically, the "condition" tag (Line 8) is used to write Boolean logic expressions, serving as the core of the rules to determine whether a file is considered a match. The sequence of strings defined under the "strings" tag (Line 5) is the basis for rule detection. By cleverly combining these strings in the "condition" section, we can construct complex and precise pattern-matching rules. We set the elements of the tuple under the "strings" tag, combined them in "condition" to form rules, and then scanned the extracted so files to match our fingerprints.

```
1 //Check emulation through system property
2 rule check_build_description_native:anti_vm
      property native
3 {meta:
4     description="ro.build.description check"
5  strings:
6     $prop="ro.build.description"
7     $str_1="release-keys"
8  condition:
9     ($prop) and $str_1}
```

Listing 1: Example of Fingerprint.

C. Test of Online Automated Dynamic Analysis Platforms

To evaluate the anti-evasion capabilities of current online app dynamic analysis platforms, we developed a probe app integrated with a comprehensive suite of evasion techniques we collected. Our main goal is to verify whether this app can accurately identify these platforms' dynamic analysis behaviors through one or more evasion techniques.

The criteria for selecting analysis platforms are 1) the platform claims to support APK file analysis; 2) the platform claims to support dynamic analysis; and 3) the availability of the service for free use[3] (i.e., allowing direct upload of samples). Both academic and business platforms are considered.

Following these criteria, 11 online dynamic analysis platforms were chosen for our experiments, including SandDroid [19], Triage [6], VirusTotal [22], QAX Threat Intelligence [18], MOBISEC [15], Koodus [14], MobSF [5], Hybrid Analysis (CrowdStrike Falcon Sandbox) [13], Tencent Habo [21], AppAudit [10], and Sixo [20].

Furthermore, our probe app does not contain malicious content, ensuring it poses no risk to the analysis platforms. Based on the reviewed evasion techniques (as summarized in Section II-B), it only gathers data pertinent to dynamic analysis behaviors and environment features. Then, if the analysis platform is equipped with a visual interface, our probe app will directly display the collected data on the interface; if the platform does not support visual displays, the probe app will send the data to our HTTP server hosted in the cloud for more in-depth subsequent analysis.

IV. Measurements and Findings

RQ1. How do evasion techniques against dynamic analysis deploy in the wild?

Based on the static analysis method described in Section III-B, we conducted a large-scale detection of evasion techniques against dynamic analysis used by Google Play apps to investigate the deployment status of these techniques in the wild. We provided a comprehensive overview of the deployment status from two dimensions: 1) the types of evasion techniques deployed in apps and 2) the locations of evasion techniques deployed in apps. Further, we delved into the possible reasons behind the deployment status.

Types of Techniques Deployed. We measured the number of apps deploying each evasion technique summarized in Section II-B, as shown in Table I. We found that evasion techniques proposed in the literature are frequently deployed in real-world apps, with their uses exceeding 80% (87,038 / 108,099). The ED5 technique is the most widespread, with

[3]Due to the limitation of research budget, some paid platforms and premium features for dynamic analysis were not covered in this study, such as 360 Sandbox Cloud [8], App-Ray [9], Appknox [11], Nanminglihuo [17], and WeTest [24].

the use of 83.4% (72,574 / 87,038). However, our further investigation into the reason revealed that the popularity of ED5 is mainly because it is frequently used for collecting user information in practice, not for evading dynamic analysis. For example, as demonstrated in Listing 2, the app checks whether an SMS is empty before retrieving its details. Since the SMS belongs to the trace of user usage in real devices, we detected that this app used ED5 (traces of usage).

```
 //Determine whether the message is empty
     and then access the info.
Cursor cursor2 = contentResolver.query(Uri.
    parse("content://sms/inbox"), ...);
if (cursor2 != null) {
    try {if (cursor2.moveToFirst()) {...}
    }catch(...){...}}
```

Listing 2: Example of accessing SMS messages.

This observation made us question whether the techniques proposed for evading dynamic analysis are actually used for this purpose. In other words, do these techniques match their intended use as stated in the literature? To clarify this, we manually analyzed apps with evasion techniques to see if they really aim to evade dynamic analysis. We randomly selected 50 APK files for each evasion technique and closely inspected their implementations, such as evaluating the correlation between the names of the methods that implement the evasion techniques and the evasion action and checking if multiple evasion techniques were used in the same method.

Our manual analysis results were listed in the third row of Table I, which shows that ED2, ED3, ED5, and TT2 are rarely used for evading dynamic analysis in practice. Their primary usage is collecting and processing device or user information. For ED4, ED6, and AH, some of their usages are not for evading dynamic analysis as well. The ping test in ED4 is used to check network connectivity, and the Modifier.isNative method in AH is typically used to verify whether a specific method is implemented as Native code. Moreover, apps determine the presence of a navigation bar by checking the qemu.hw.mainkeys property in ED6 in order to perform the subsequent operations about the navigation bar.

Further, we explored the possible reasons for this phenomenon. As stated in Section II-B, the ED2, ED3, and ED5 techniques evade dynamic analysis by detecting whether the executing environment is virtual. For ED2 and ED3, which rely on hardware and software component information, the detection results may be affected by hardware and software compatibility differences among devices, making the results inaccurate. For ED5, it utilizes the traces of user usage in real devices. This evasion technique may not work because virtual environments can easily simulate the traces of usage. The TT2 technique and the Modifier.isNative method in AH technique need to detect whether Java methods have been converted to Native methods. The implementation of this detection is complex and time-consuming, which may affect the performance of the app. As a result, these evasion techniques are hardly used for evading dynamic analysis in practice under the consideration of app performance, technical reliability, and difficulty in technique implementation.

TABLE II: Evasion techniques locations deployed in apps.

Location	Usage Status	App Count	Percentage
Local	Used	56,199	52.0%
Third-party Library	Used	37,272	34.5%
Third-party Library	Unused	80,376	74.4%

After excluding evasion techniques rarely used for anti-dynamic analysis in real-world apps, the most common technique is ED1 (61.4%), followed by AD2 (43.5%) and AD1 (29.5%). The least frequently used technique is AH (1.2%). The high deployment rate of ED1 may be attributed to its ability to retrieve system property information easily through simple API calls without significantly affecting app performance. The AD1 and AD2 techniques are prevalent probably because debugging techniques are quite common, and the AD techniques are simple to implement. The low prevalence of the AH technique might relate to its high deployment difficulty and the rarity of hooking techniques, making AH mainly found in apps with stringent security requirements.

Finding 1. Techniques proposed in the literature for evading dynamic analysis are not always used for this purpose in real-world apps. After identifying the techniques actually used for evasion, we measured that 68.1% (73,607 / 108,099) of apps have deployed evasion techniques. These apps predominantly rely on ED1 and AD techniques to evade dynamic analysis because these methods are easy to implement and widely reliable and have little affection for app performance.

Locations of Techniques Deployed. Additionally, we investigated the locations of evasion technique deployments in apps. Firstly, we analyzed whether the evasion techniques were implemented in the app's local code (code built by the app developers themselves) or in the third-party library code, and if the latter, whether they can be invoked by the app's local code. As mentioned in Section III-B, the implementations of evasion techniques in the app's local code or the third-party code invoked by the app's local code indicate that the app indeed used such techniques. According to the results shown in Table II, a higher proportion (52.0%) of evasion techniques were found in the app's local code. On the other hand, the number of apps containing evasion technique implementations within third-party libraries reached 81,418 (75.2%). However, most (74.4%) of them did not use these techniques (i.e., the corresponding implementations were not invoked by the app's local code).

Further, as shown in Table III, we identified the top five third-party libraries with the most evasion techniques and found that none of them focused on evading dynamic analysis. The evasion techniques deployed in third-party libraries are primarily for the specific needs of the libraries themselves. Therefore, app developers seem to prefer implementing evasion techniques within the app's local code.

Secondly, based on the code layers at which the evasion technique is located, we categorized the evasion techniques into

TABLE III: Top 5 third-party libraries with evasion capability.

Third-party Library	Type	App Count
Lcom/google/android/gms	Development Aid	72,045
Lcom/facebook	Social Network	19,554
Lcom/google/firebase	Development Aid	8,216
Lorg/apache/cordova	Development Aid	6,940
Lio/fabric/sdk/android	Development Aid	6,793

two layers: the Java / Kotlin layer and the Native layer. The fourth row of Table I shows that evasion techniques are mainly concentrated in the Java / Kotlin layer, accounting for 64.0%. On the contrary, the Native layer only accounts for 11.7%. The proportion of most evasion techniques in the Native layer is much lower than that of corresponding techniques in the Java / Kotlin layer, such as ED1 and AD2. This phenomenon may be explained by the fact that the Android development environment is mainly designed for Java / Kotlin, and developers are more familiar with and proficient in these languages.

Finding 2. App developers tend to deploy evasion techniques directly in their local code rather than using third-party libraries and prefer implementing evasion techniques in Java / Kotlin code. Most third-party libraries containing numerous evasion techniques are not designed for evading dynamic analysis but to meet their own evasion needs.

Answers to RQ1

68.1% of apps have deployed evasion techniques, with ED1 being the most commonly used. Moreover, app developers tend to implement evasion techniques in Java / Kotlin and within the app's local code.

RQ2. What are the differences between evasion techniques deployed by benign and malicious apps?

As introduced in Section I, benign apps strive to evade analysis by potential attackers to protect intellectual property rights and prevent unauthorized copying. Whereas malicious apps utilize evasion techniques to avoid being identified by analysis platforms. These two kinds of apps have different intentions to evade dynamic analysis. Hence, we devoted particular attention to the discrepancies in evasion tactics deployed by benign and malicious apps in the aspects of the deployment status and evasion behaviors after identifying the dynamic analysis scenarios.

Differences in Deployment Status. In our analysis, as mentioned in Section III-A, we treated the collected Google Play apps as benign apps, and the deployment status of dynamic evasion techniques in them was analyzed in RQ1. Similarly, we examined the deployment status of evasion techniques in malicious apps. The results are shown in Table IV. Contrary to our expectations, only 15.8% (1,848 / 11,730) of malicious apps utilized such techniques, in comparison to 68.1% for benign apps. This unexpectedly lower incidence of dynamic analysis evasion in malicious apps compared to benign ones suggests a high need for benign apps to protect their code and data. For example, banking apps often involve sensitive data and place a higher emphasis on protective measures. On the other hand, malicious app developers may be good at hiding their behaviors and avoiding the detection of evasion techniques they deploy.

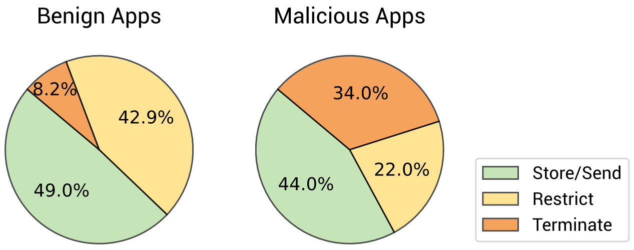

Fig. 2: Statistics on evasion behaviors of apps.

As for the types of evasion techniques deployed, the most popular evasion technique used in malicious apps is also ED1, but compared to benign apps, which often use the `Build` class in ED1, malicious apps are more inclined to use the `TelephonyManager.getDeviceId` method in ED1. On the other hand, the least commonly used technique in benign apps is AH. While in malicious apps, it is TT1 (Monkey detection). It reflects that most malware does not prioritize targeting users of the Monkey testing tool, as their primary design is to operate covertly and avoid detection. Specifically, detecting the uncommon Monkey testing tool could increase the risk of the malware being discovered.

Finding 1. Malicious apps generally deploy evasion techniques less frequently than benign apps. Both kinds of apps use the ED1 technique most commonly, but the specific technique implementations are different. The least commonly used technique is AH in benign apps and TT1 in malicious apps.

Differences in Evasion Behaviors. Furthermore, we investigated the differences between benign and malicious apps in their evasion behavior patterns after identifying that they are being dynamically analyzed. Given the complexity of distinguishing specific behaviors, static analysis often falls short of accurately discerning developers' intentions. Therefore, we manually analyzed the behaviors after evading dynamic analysis in 50 benign apps and 50 malicious apps, respectively.

We found that the evasion behaviors of an app can be classified into three principal patterns as follows:

- *Store or send detection information*: the app locally stores the detection information that indicates whether it is being dynamically analyzed. Or it sends this information to a remote server.
- *Restrict certain functionalities*: the app changes the execution path to restrict the implementation of original functionalities.
- *Terminate the running*: the app kills the process or throws an exception to terminate its running.

As shown in Figure 2, of the three evasion behavior patterns, both benign and malicious apps prefer storing or sending detection information. The difference is that benign apps tend

TABLE IV: Statistics on malicious apps using evasion techniques.

Evasion Techniques	**ED1**	**ED2**	**ED3**	**ED4**	**ED5**	**ED6**	**AD1**	**AD2**	**AH**	**TT1**	**TT2**	**All***
All	1,403	1,238	320	30	4,134	302	75	447	48	11	2	4,877
TiE*	45/50	0/50	0/50	18/30	0/50	32/50	50/50	50/50	48/48†	11/11	0/2†	-
Java / Kotlin	1,148	-	-	18	-	82	59	419	25	11	-	1,475
Native	300	-	-	0	-	158	16	32	26	-	-	457
All (adjusted)	1,403	-	-	18	-	239	75	447	48	11	-	1,848
Percentage	12.0%	-	-	0.2%	-	2.0%	0.6%	3.8%	0.4%	0.1%	-	15.8%

$\star$: **T**echniques **i**ndeed **E**vading Dynamic Analysis in Practice
†: Less than 50 malicious apps are using AH / TT2 in our APK dataset.
*: Some apps implement the evasion actions in both Java/Kotlin and Native code. Therefore, the [All] amount is not [Java/Kotlin] + [Native] directly.

TABLE V: Experimental results for online dynamic analysis services.

Anslysis Platform	**ED1**	**ED2**	**ED3**	**ED4**	**ED5**	**ED6**	**ED7**	**AD1**	**AD2**	**AH**	**TT1**	**TT2**	**TT3**
SandDroid [19]	✓	✓	✓	✓	✓	✓	−	×	✓	×	×	−	×
Triage [6]	✓	✓	✓	✓	✓	✓	✓	×	✓	✓	×	✓	×
VirusTotal [22]	✓	✓	✓	✓	✓	✓	✓	×	✓	✓	×	✓	✓
QAX [18]	✓	✓	✓	✓	✓	✓	✓	×	✓	✓	×	✓	×
MOBISEC [15]	✓	✓	✓	✓	✓	✓	✓	×	×	×	×	−	×
Koodus [14]	✓	✓	✓	✓	✓	✓	✓	×	✓	×	×	✓	×
MobSF [5]	✓	✓	✓	✓	✓	✓	✓	×	✓	✓	×	−	×

✓: This technique can effectively identify dynamic analysis platforms.
−: This technique has not been triggered.
×: This technique can not identify dynamic analysis platforms.

to store information locally, probably because benign apps try to protect their privacy by minimizing external user data transmission. On the contrary, malicious apps are apt to send information to a remote server. It is possibly because malicious apps want to conceal their negative intentions by performing malicious decision-making processes on the remote server based on the detected information. Additionally, malicious apps are more inclined to terminate immediately after identifying being dynamically analyzed, perhaps in order to hide their subsequent malicious behaviors.

Finding 2. After identifying being dynamically analyzed, apps may modify their behavior, terminate execution, or access detection information to evade analysis. Malicious apps are more likely to terminate execution or execute malicious logic on a remote server to hide their malicious behaviors, whereas benign apps are more likely to continue running normally without disruption.

Answers to RQ2

The proportion of evasion techniques deployed in malicious apps is lower compared to benign ones, at only 15.8%. Compared to benign apps, malicious apps' evasion behaviors are often more cautious, such as actively hindering further operations.

RQ3. Can existing evasion techniques effectively evade the detection of online automated dynamic analysis platforms?

In this part, we discuss the effectiveness of evasion techniques and the anti-evasion performance of online analysis platforms. As mentioned in Section III-C, we developed a probe app and uploaded it to 11 online malware detection platforms that support dynamic analysis. Specifically, we incorporated all known implementations of each evasion technique. If any technique implementation successfully evades the analysis platforms, we marked this evasion technique to be effective.

During the experiment, we were unable to obtain detection results from four platforms, including Hybrid Analysis, Tencent Habo [21], AppAudit [10], and Sixo [20], because they neither sent responses to our server nor provided a visual interface to view the data. Possible reasons are that these platforms limit the app's network access or do not actually perform dynamic analysis. However, due to the black-box nature of online platform operations, pinpointing the precise reason remains challenging. Finally, data from seven platforms were analyzed, including SandDroid, Triage, VirusTotal, QAX Threat Intelligence, MOBISEC, Koodus, and MobSF. The cumulative results are presented in Table V.

Effectiveness of Evasion Techniques. According to Table V, most evasion techniques demonstrated efficacy in detecting automated dynamic analysis environments. Notably, emulator detection techniques (ED1 ~ ED7) were capable of identifying nearly all platforms, indicating that the majority of these platforms are developed based on emulator environments. However, the techniques for detecting debugger (AD1) and Monkey (TT1) did not succeed in evasion. This may be due to the absence of Monkey and debugger deployment on these platforms.

Finding 1. Except for AD1 and TT1, the evasion techniques mentioned in the literature are able to evade most online dynamic analysis platforms. It shows that the anti-evasion capability of the current dynamic analysis platform needs to be strengthened.

Anti-Evasion Strategies of Platforms. Although most types of evasion techniques can effectively evade online dynamic analysis platforms, our analysis results show that some malware detection platforms have actually deployed anti-evasion strategies against specific implementations. For instance, Trige, VirusTotal, Koodous, and MOBISEC have either obscured their device information or utilized real devices in their analysis processes. These anti-evasion measures can mainly be divided into the following three categories:

System Property. Analysis platforms simulate the system property of real devices or use the real devices as test devices. For instance, we queried the system properties from the Koodus platform (as shown in Listing 3) and found it simulates or really uses a OnePlus 8 Pro phone. Nevertheless, Koodus and some other platforms (e.g., Triage and VirusTotal) operate on multi-device analysis frameworks and have not entirely succeeded in camouflaging the specifics of each device. MOBISEC, built on a single device, exhibits an imperfect disguise, exemplified by a phone number beginning with 1555. Therefore, these platforms still cannot resist the ED1 technique.

```
MODEL: OnePlus8Pro
BRAND: OnePlus
DEVICE: OnePlus8Pro
TAGS: release-keys
FINGERPRINT: OnePlus/IN2023/OnePlus8Pro:11/
    QKR1.191246.002/2006701342:user/release-
    keys
```

Listing 3: System properties on Koodus.

Traces of Usage. Analysis platforms fake the traces of usage. Using the Triage platform as an example, there exists a device profile that nearly encapsulates all usage information, as detailed in Listing 4. This profile encompasses fabricated contacts, app packages, and even geographical coordinates (latitude and longitude). However, due to the incompleteness of the usage information, we still can evade Triage using the ED5 technique.

```
Contacts: PhoneNumber:88001007337;
Latitude: 9.9139368
Longitude: 78.0911564
PackageName: com.google.android.apps.
    inputmethod.hindi
```

Listing 4: Traces of usage on Triage.

Performance. Analysis platforms simulate the performance of real devices or use real devices to test. For example, evidence suggests that Triage likely employed at least one physical device for testing, as indicated by a CPU computation time of 15 ms. In contrast, emulators typically demonstrate computation times ranging from 2 to 3 ms.

Finding 2. Although some online dynamic analysis platforms have deployed certain anti-evasion measures, which mainly include simulating real attribute values, usage traces, and performance characteristics. However, these measures are not deployed comprehensively. Therefore, when an app is equipped with sophisticated evasion techniques, these platforms can still be effectively evaded.

Answers to RQ3

Evasion techniques can effectively detect automated analysis environments, particularly through emulator detection methods. Analysis platforms have implemented counter-evasion measures, but the implementations are not enough.

V. Suggestions

Android Apps. The techniques that apps typically use to evade dynamic analysis, such as ED1, mostly derive from basic techniques proposed in literature years ago. Their widespread use may be due to their ease of implementation and integration. However, as dynamic analysis platforms have gradually developed countermeasures against these older techniques, we recommend that benign apps deploy various evasion techniques to enhance their ability to resist these countermeasures.

Automated Analysis Platforms. Existing dynamic analysis platforms are almost defenseless, even against the most basic evasion techniques. Take Koodus as an example. Although it attempts to deploy anti-evasion mechanisms, its use of multiple devices without ensuring comprehensive deployment makes it vulnerable. Malicious app developers can easily make local decisions and take appropriate actions when detecting a virtual environment. This situation highlights that the current technological abilities of online analysis platforms are nearly ineffective against the evasion attempts of malicious entities. Given the differences in how benign and malicious apps use evasion techniques, platforms can design specific rules to determine whether an app's evasion behavior is motivated by malicious intent. For instance, platforms could assess the behavior patterns after evasion. Fine-grained judgment allows analysis platforms not only to identify malicious activities accurately but also to minimize undue harm to legitimate apps.

VI. Threats to Validity

In this section, the potential threats to the validity of the study are discussed.

Empirical Analysis Limitations. Some of the techniques proposed in the literature, while usable for evading dynamic analysis, are not always intended for this purpose in practice. Therefore, we adopted a strict manual analysis approach to exclude the techniques rarely used for evasion in real-world apps. However, there may be apps in our APK dataset whose behaviors do not align with the empirical conclusions drawn from manual analysis, which could impact the accuracy of the research results.

Static Analysis Limitations We observed that in most cases, evasion techniques were not combined with anti-static analysis techniques such as code obfuscation. However, there is still a possibility that evasion techniques in a few apps may be obscured by anti-static analysis techniques, which could lead to deviations in our static analysis results.

Third-party Tools Error. This study relied on LiteRadar to extract third-party libraries in apps. Although LiteRadar represents an advanced tool in its domain, its detection capabilities of third-party libraries may not achieve complete accuracy.

VII. Related Work

A lot of publications have extensively explored research on evasion techniques against dynamic analysis in apps. These publications can be principally divided into three categories: the proposal of evasion techniques, the proposal of anti-evasion techniques, and the detection of evasion techniques in apps.

Evasion Methods. The high-level idea of evasion techniques is to identify the differences between the real user-end and app analysis environments. Vidas et al. [49] first proposed a method to detect Android runtime analysis systems. They detect virtualized dynamic analysis platforms by analyzing differences in behavior, performance, and system components caused by design choices. Petsas et al. [44] proposed a technique that utilizes static attributes, dynamic sensor information, and complex information related to the Android virtual machine to evade dynamic analysis. Maier et al. [41] developed a tool called Sand-Finger, which can identify fingerprints about Android analysis systems. Jing et al. [38] proposed a framework called Morpheus, which analyzes and compares artifacts retrieved from Android emulators and real devices to generate heuristic detection emulator methods. Diao et al. [33] proposed a new way to evade runtime analysis by detecting differences in interaction patterns of machines and human users. Costamagna et al. [31] have proposed detecting emulation environments based on traces of device usage, such as obtaining contacts and sms. Wan et al. [50] analyzed several open-source Android hooking tools, which debug apps through various hook points. They summarized these points and effectively implemented anti-hooking by verifying their integrity at runtime. Sihag et al. [46] investigated and summarized the Android malware enhancement techniques and conducted a detailed classification, including part of dynamic analysis evasion technology. Kondracki et al. [39] developed an "environment-aware" sandbox detection technique using an Android app to collect API statistics on wear-and-tear artifacts and hardware components. They compared data from both sandbox environments and real devices and then built a machine-learning classifier to distinguish between them.

Anti-Evasion Methods. The high-level idea of anti-evasion techniques is to pretend the current app analysis environment is a real user's phone. Mutti et al. [42] proposed BareDroid for bare metal analysis of Android apps. It can quickly revert a real device to a clean snapshot and remain imperceptible to malware. Rasthofer et al. [45] proposed HARVESTER, which combines static and dynamic analysis to extract and execute key code snippets, bypassing malware's environmental detection mechanisms. Ning et al. [43] developed NINJA, a malware analysis framework using TrustZone technology for transparent trace analysis, minimizing system performance impact. Bordoni et al. [29] proposed Mirage, the first Android malware sandbox architecture. It uses four advanced techniques to prevent evasion. Firstly, it collects data from real devices to create realistic simulations. Secondly, it expertly hooks and alters the Android API return values to mirror those of real devices. Thirdly, Mirage replays event streams from actual devices to simulate their behavior accurately. Lastly, it ensures the consistency of false data injected into the emulator and records the analysis process to discover and respond to evasion techniques. Song et al. [47] proposed VPBox, a new Android operating system-level sandbox framework implemented based on container virtualization. VPBox achieves transparent and covert dynamic analysis of Android apps by combining kernel-level and user-level device virtualization techniques, along with device attribute and SELinux customization. Faghihi et al. [35] introduce CamoDroid, which simulates real device data, sensors, user input, static and network characteristics and hides the existence of the analysis environment. Hayyan et al. [36] proposed the Maaker framework, which uses model-driven engineering to put humans in the loop and then uses human knowledge to deal with different evasion behaviors. Cui et al. [32] developed a framework called DroidHook that can customize the monitoring API set and run on real devices.

Evasion Methods Detection Afonso et al. [26] proposed a method called Lumus, which effectively identifies app samples attempting to evade detection by comparing the execution trajectories of malicious apps in bare-metal (i.e., physical devices) and simulated environments. Berlato et al. [28] conducted a large-scale study of Android apps to quantify the actual app of anti-debugging and anti-tampering protection techniques.

VIII. Conclusion

This study systematically investigates the evasion techniques used by Android apps against dynamic analysis. We designed three key research questions to analyze the real-world use of evasion techniques, compare their use between benign and malicious apps, and evaluate the effectiveness of current online dynamic analysis platforms. Our large-scale empirical analysis, covering 108,099 benign apps, 11,730 malicious apps, and 11 online automated dynamic analysis platforms, provided detailed insights into the current state of Android dynamic evasion techniques. Our findings show that many apps use evasion techniques, with benign apps using them more frequently than malicious ones. Additionally, we found that existing dynamic analysis platforms often fail to counter these evasion methods effectively. To address these challenges, we offered recommendations for app and platform developers to help combat malicious activities and protect legitimate apps. Our research enhances the understanding of Android anti-analysis techniques and offers valuable directions for future research.

Acknowledgements

This work was partially supported by Taishan Young Scholar Program of Shandong Province, China (Grant No. tsqn202211001), Shandong Provincial Natural Science Foundation (Grant No. ZR2023MF043), and Xiaomi Young Talents Program.

References

[1] (2020) LiteRadar. [Online]. Available: https://github.com/pkumza/LiteRadar

[2] (2023) AndroZoo. [Online]. Available: https://androzoo.uni.lu/

[3] (2023) Data Flow Analysis. [Online]. Available: https://en.wikipedia.org/wiki/Data-flow_analysis

[4] (2023) dex2jar. [Online]. Available: https://github.com/pxb1988/dex2jar

[5] (2023) MobSF. [Online]. Available: https://github.com/MobSF/Mobile-Security-Framework-MobSF

[6] (2023) Recorded Future Triage. [Online]. Available: https://tria.ge/login?return_to=%2Fdashboard

[7] (2023) Yara. [Online]. Available: https://yara.readthedocs.io/en/stable/writingrules.html

[8] (2024) 360 Sandbox Cloud. [Online]. Available: https://ata.360.net/

[9] (2024) App-Ray. [Online]. Available: http://app-ray.co/

[10] (2024) AppAudit. [Online]. Available: http://appaudit.io/

[11] (2024) Appknox. [Online]. Available: https://www.appknox.com/

[12] (2024) Google Play. [Online]. Available: https://play.google.com/store/games?device=windows

[13] (2024) Hybrid Analysis. [Online]. Available: https://www.hybrid-analysis.com/

[14] (2024) Koodus. [Online]. Available: https://koodous.com/

[15] (2024) MOBISEC. [Online]. Available: https://challs.reyammer.io/apks

[16] (2024) Monkey. [Online]. Available: https://developer.android.com/studio/test/other-testing-tools/monkey

[17] (2024) Nanminglihuo. [Online]. Available: https://www.zhihuaspace.cn:8888/

[18] (2024) QAX Threat Intelligence. [Online]. Available: https://ti.qianxin.com/

[19] (2024) SandDroid. [Online]. Available: https://sanddroid.xjtu.edu.cn/

[20] (2024) Sixo Online APK Analyzer. [Online]. Available: https://www.sisik.eu/apk-tool

[21] (2024) Tecent Habo. [Online]. Available: https://habo.qq.com/

[22] (2024) VirusTotal. [Online]. Available: https://developers.virustotal.com/docs/docs-in-house-sandboxes

[23] (2024) WALA. [Online]. Available: https://github.com/wala/WALA

[24] (2024) WeTest. [Online]. Available: https://wetest.qq.com/products/application-safe-testing

[25] A. Afianian, S. Niksefat, B. Sadeghiyan, and D. Baptiste, "Malware Dynamic Analysis Evasion Techniques: A Survey," *ACM Computing Surveys*, vol. 52, no. 6, pp. 126:1–126:28, 2020.

[26] V. M. Afonso, A. Kalysch, T. Müller, D. Oliveira, A. Grégio, and P. L. de Geus, "Lumus: Dynamically Uncovering Evasive Android Applications," in *Proceedings of the 21st Information Security Conference (ISC), Guildford, UK, September 9-12, 2018*, 2018.

[27] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. D. McDaniel, "Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Edinburgh, United Kingdom, June 09 - 11, 2014*, 2014.

[28] S. Berlato and M. Ceccato, "A Large-scale Study on the Adoption of Anti-debugging and Anti-tampering Protections in Android Apps," *Journal of Information Security and Applications*, vol. 52, p. 102463, 2020.

[29] L. Bordoni, M. Conti, and R. Spolaor, "Mirage: Toward a Stealthier and Modular Malware Analysis Sandbox for Android," in *Proceedings of the 22nd European Symposium on Research in Computer Security (ESORICS), Oslo, Norway, September 11-15, 2017*, 2017.

[30] H. Cho, J. Lim, H. Kim, and J. H. Yi, "Anti-debugging Scheme for Protecting Mobile Apps on Android Platform," *The Journal of Supercomputing*, vol. 72, pp. 232–246, 2016.

[31] V. Costamagna, C. Zheng, and H. Huang, "Identifying and Evading Android Sandbox Through Usage-Profile Based Fingerprints," in *Proceedings of the 1st workshop on radical and experiential security (RESEC), Incheon, Korea, June 4-8, 2018*, 2018.

[32] Y. Cui, Y. Sun, and Z. Lin, "DroidHook: A Novel API-hook based Android Malware Dynamic Analysis Sandbox," *Automated Software Engineering*, vol. 30, p. 10, 2023.

[33] W. Diao, X. Liu, Z. Li, and K. Zhang, "Evading android runtime analysis through detecting programmed interactions," in *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec), Darmstadt, Germany, July 18-22, 2016*, 2016.

[34] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang, "Understanding Android Obfuscation Techniques: A Large-Scale Investigation in the Wild," in *Proceedings of the 14th International Conference on Security and Privacy in Communication Networks (SecureComm), Singapore, August 8-10, 2018*, 2018.

[35] F. Faghihi, M. Zulkernine, and S. H. H. Ding, "CamoDroid: An Android Application Analysis Environment Resilient against Sandbox Evasion," *Journal of Systems Architecture*, vol. 125, p. 102452, 2022.

[36] H. Hasan, B. T. Ladani, and B. Zamani, "Maaker: A Framework for Detecting and Defeating Evasion Techniques in Android Malware," *Journal of Information Security and Applications*, vol. 78, p. 103617, 2023.

[37] R. Hay, O. Tripp, and M. Pistoia, "Dynamic detection of inter-application communication vulnerabilities in android," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, 2015.

[38] Y. Jing, Z. Zhao, G. Ahn, and H. Hu, "Morpheus: Automatically Generating Heuristics to Detect Android Emulators," in *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC), New Orleans, LA, USA, December 8-12, 2014*, 2014.

[39] B. Kondracki, B. A. Azad, N. Miramirkhani, and N. Nikiforakis, "The Droid is in the Details: Environment-aware Evasion of Android Sandboxes," in *Proceedings of the 29th Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, April 24-28, 2022*, 2022.

[40] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and Y. L. Traon, "Static analysis of android apps: A systematic literature review," *Information and Software Technology*, vol. 88, pp. 67–95, 2017.

[41] D. C. Maier, T. Müller, and M. Protsenko, "Divide-and-Conquer: Why Android Malware Cannot Be Stopped," in *Proceedings of the 9th International Conference on Availability, Reliability and Security (ARES), Fribourg, Switzerland, September 8-12, 2014*, 2014.

[42] S. Mutti, Y. Fratantonio, A. Bianchi, L. Invernizzi, J. Corbetta, D. Kirat, C. Kruegel, and G. Vigna, "BareDroid: Large-Scale Analysis of Android Apps on Real Devices," in *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC), Los Angeles, CA, USA, December 7-11, 2015*, 2015.

[43] Z. Ning and F. Zhang, "Ninja: Towards Transparent Tracing and Debugging on ARM," in *Proceedings of the 26th USENIX Security Symposium (USENIX-Sec), Vancouver, BC, Canada, August 16-18, 2017*, 2017.

[44] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Rage against the Virtual Machine: Hindering Dynamic Analysis of Android Malware," in *Proceedings of the 7th European Workshop on System Security (EuroSec), Amsterdam, The Netherlands, April 13, 2014*, 2014.

[45] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques," in *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 21-24, 2016*, 2016.

[46] V. Sihag, M. Vardhan, and P. Singh, "A Survey of Android Application and Malware Hardening," *Computer Science Review*, vol. 39, p. 100365, 2021.

[47] W. Song, J. Ming, L. Jiang, Y. Xiang, X. Pan, J. Fu, and G. Peng, "Towards Transparent and Stealthy Android OS Sandboxing via Customizable Container-Based Virtualization," in *Proceedings of the 28th ACM SIGSAC Conference on Computer and Communications Security (CCS), Virtual Event, Republic of Korea, November 15 - 19, 2021*, 2021.

[48] X. Sun, X. Chen, L. Li, H. Cai, J. Grundy, J. Samhi, T. F. Bissyandé, and J. Klein, "Demystifying Hidden Sensitive Operations in Android Apps," *ACM Transactions on Software Engineering and Methodology*, vol. 32, pp. 50:1–50:30, 2023.

[49] T. Vidas and N. Christin, "Evading Android Runtime Analysis via Sandbox Detection," in *Proceedings of the 9th ACM Asia Conference on Computer and Communications Security (ASIACCS), Kyoto, Japan, June 03 - 06, 2014*, 2014.

[50] J. Wan, M. Zulkernine, and C. Liem, "A Dynamic App Anti-Debugging Approach on Android ART Runtime," in *Proceedings of the 3rd Cyber Science and Technology Congress (CyberSciTech), Athens, Greece, August 12-15, 2018*, 2018.