2023

A Survey and Evaluation of Android-Based Malware Evasion Techniques and Detection Frameworks

Parvez Faruki
Government of Gujarat

Rhati Bhan
Indian Institute of Technology

Vinesh Jain
Engineering College Ajmer

Sajal Bhatia
Sacred Heart University

Nour El Madhoun
LISITE Laboratory

See next page for additional authors

Recommended Citation

Faruki, P., Bhan, R., Jain, V., Bhatia, S., El Madhoun, N., & Pamula, R. (2023). A survey and evaluation of android-based malware evasion techniques and detection frameworks. *Information, 14*(7), 374. Doi: 10.3390/info14070374

Authors
Parvez Faruki, Rhati Bhan, Vinesh Jain, Sajal Bhatia, Nour El Madhoun, and Rajendra Pamula

Review

A Survey and Evaluation of Android-Based Malware Evasion Techniques and Detection Frameworks

Parvez Faruki [1], Rati Bhan [2], Vinesh Jain [3], Sajal Bhatia [4,*], Nour El Madhoun [5] and Rajendra Pamula [2]

[1] Department of Technical Education, Government of Gujarat, Gandhinagar 382010, India
[2] Department of Computer Science and Engineering, Indian Institute of Technology (ISM), Dhanbad 826004, India; rajendra@iitism.ac.in (R.P.)
[3] Department of Computer Science and Engineering, Engineering College Ajmer, Ajmer 305001, India
[4] School of Computer Science and Engineering, Sacred Heart University, Fairfield, CO 06825, USA
[5] LISITE Laboratory, ISEP, 10 Rue de Vanves, 92130 Issy-les-Moulineaux, France; nour.el-madhoun@isep.fr
* Correspondence: bhatias@sacredheart.edu

Abstract: Android platform security is an active area of research where malware detection techniques continuously evolve to identify novel malware and improve the timely and accurate detection of existing malware. Adversaries are constantly in charge of employing innovative techniques to avoid or prolong malware detection effectively. Past studies have shown that malware detection systems are susceptible to evasion attacks where adversaries can successfully bypass the existing security defenses and deliver the malware to the target system without being detected. The evolution of escape-resistant systems is an open research problem. This paper presents a detailed taxonomy and evaluation of Android-based malware evasion techniques deployed to circumvent malware detection. The study characterizes such evasion techniques into two broad categories, polymorphism and metamorphism, and analyses techniques used for stealth malware detection based on the malware's unique characteristics. Furthermore, the article also presents a qualitative and systematic comparison of evasion detection frameworks and their detection methodologies for Android-based malware. Finally, the survey discusses open-ended questions and potential future directions for continued research in mobile malware detection.

Keywords: android malware; evasion techniques; code obfuscation; code transformation; reflection; dynamic code loading; mobile security; machine learning

check for updates

Citation: Faruki, P.; Bhan, R.; Jain, V.; Bhatia, S.; El Madhoun, N.; Pamula, R. A Survey and Evaluation of Android-Based Malware Evasion Techniques and Detection Frameworks. *Information* **2023**, *14*, 374. https://doi.org/10.3390/info14070374

Academic Editors: Alexander Perez-Pons, Upadhyay Himanshu and Tushar Bhardwaj

Received: 11 April 2023
Revised: 12 May 2023
Accepted: 24 May 2023
Published: 30 June 2023

1. Introduction

Smart-device security and privacy are essential since adversarial attacks have increasingly stolen and misused confidential user information via stealth apps; the existing anti-malware solutions deploy advanced techniques for detecting evasive malware [1]. However, spyware [2], botnets, premium-rate SMS Trojans, banking Trojans, aggressive advertising [3], and privilege escalation techniques have successfully attacked the official market Google Play and other third-party app stores at regular intervals. Furthermore, malware variants increased by 54% from 2017 to 2018, triggered by the emergence of novel techniques [4].

The adversaries employ clever techniques and steal confidential user information. For example, malware authors use privileged ways and gain access to smartphone apps. This includes stealing mobile contacts and text messages, recording calls, and dialing premium-rate numbers, thus incurring a monetary loss; further, attackers gain root privileges, steal confidential banking details and track user locations via GPS through advanced attack techniques. Java language is used for Android app development, with the capability to protect commercial software from piracy. However, cybercriminals deploy persistent attacks via anti-malware evasion tools, including code obfuscation, polymorphism, and

Java reflections; some techniques are popular for genuine software protection. These attacks are mainly aimed at thwarting stealth attack detection.

Evasion techniques are aimed at encryption, polymorphism, code transformation or code-obfuscation, and package renaming that are a part of attacks [5]. In addition, obfuscation methods transform the existing malware and recreate variants to avoid detection. Cyber-criminals deploy malicious anti-malware tools to propagate persistent threats [6]; thus, the accuracy and performance of malware detection frameworks deteriorate. Various studies have identified relations between evasion techniques that undermine machine learning-based anti-malware approaches [7–15]. For example, Drebin [16] obtained a detection accuracy of 94% and noted that dynamic code loading is one of the critical causes of failure. On the other hand, Elish et al. [7] claimed that malware families containing reflection API and code obfuscation could evade anti-malware and remain undetected. Similarly, Chen et al. [15] used code graph similarity to identify repackaged apps in ten seconds but could not investigate the injection of intrusive code. Thus, this increases the curiosity to explore and analyze evasion techniques to improve the malware detection and analysis approaches.

The following studies listed in Table 1 reveal that adversaries use dominant techniques such as packing, encryption, code transformation via metamorphism or polymorphism, and virtual environment detection to camouflage from antimalware. The studies of [17–20] performed an evasion techniques review; however, they did not provide a comprehensive review and robust conclusions. For example, Rastogi et al. [21] developed DroidChameleon a code transformation framework to dodge the commercial anti-malware solutions. Sufatrio et al. [19] investigated malware analysis techniques and similar strategies for eluding literary and commercial anti-malware techniques. However, they failed to mention the taxonomy of the evasion methods and evasion detection techniques. The following study fills this literature gap by presenting a systematic taxonomy of the evasion techniques and their impact that hinders anti-malware solutions [22]. Another goal is to emphasize the significance of evasion tools, techniques, impact, countermeasures, and open questions and address possible obstacles for upcoming research. The indications "$\surd$", "$\times$", or an empty cell intersecting the framework row with the evasion column identify researchers who tested their framework against certain evasions. "$\surd$" indicates that the study either tested or assumed it could detect the evasion tactic. At the same time, "$\times$" means the researcher assumed that the evasion technique bypassed their Android malware detection framework. Incomplete reports of framework evaluation studies on evasion tactics or assumptions are shown by an empty cell.

Table 1 emphasizes the contribution of this study by filling the literature gap in evasive techniques, their detection, and analysis. Here, the types of evasion techniques are abbreviated as code encryption (CE), code transformation (CT), code obfuscation (CO), package transformation (PT), and anti-emulation (AE). The table clearly shows other techniques that focus on, at most, one or two types of evasion, whereas this study tries to cover all five types mentioned above. Moreover, existing literature compared studies that only included commercial anti-malware tools, whereas this survey covers commercial and important academic contributions.

Table 1. Contributions and Comparison.

Related Study	Year	Types of Evasion Techniques Covered					Type of Evasion Tool Included	
		Code Encryption	Code Transformation	Code Obfuscation	Package Transformation	Enti-Emulation	Commercial	Academic
This study	2023	√	√	√	√	√	√	√
Faghihi et al. [6]	2022	√			√			√
Sihag et al. [23]	2021			√	√			√
Jusoh et al. [24]	2021	√	√	√	√	√	√	√

Table 1. *Cont.*

		Types of Evasion Techniques Covered					Type of Evasion Tool Included	
Related Study	**Year**	**Code Encryption**	**Code Trans-formation**	**Code Ob-fuscation**	**Package Trans-formation**	**Enti-Emulation**	**Commercial**	**Academic**
Aslan et al. [25]	2020			√				√
Razgallah et al. [26]	2020		√	√	√			√
Chen et al. [27]	2019			√	√	√		√
Parnika et al. [28]	2019				√			√
Sen et al. [29]	2018	√		√		√		√
Dai et al. [30]	2018		√		√			√
Xue et al. [31]	2017	√					√	
Tam et al. [32]	2017		√	√				
Nguyen-Vu et al. [33]	2017		√					
Preda et al. [34]	2016			√			√	
Hoffmann et al. [35]	2016			√			√	
Kim et al. [36]	2016					√		
Sufatrio et al. [19]	2015			√				
Faruki et al. [37]	2014	√	√				√	
Rastogi et al. [38]	2013		√				√	

Selection Criteria

The retrieved articles based on a search query (SQ) would not fit the paper's scope, so we needed to filter them by applying inclusion–exclusion criteria.

Inclusion criteria: The survey's scope extends over the intersection of four identified research domains (RDs).

Inclusion Criteria: The scope of the survey extends over the intersection of four identified Research Domain (RD).

1. **Apps Analysis** RD encompasses all the approaches for identifying smartphone app models for Android platform.
2. **Security Analysis** RD encompasses all approaches for detecting security flaws in extracted models.
3. **Anti-emulation transformation** RD includes the techniques involved in Anti-emulation transformation explained in Table 2.
4. **Evasion Techniques** RD covers various Evasion Techniques for Android Malware Detection shown in Figure 1.

Table 2. Anti-emulation transformation.

S.No	**Anti-Emulation Approaches**	**Technique**	**Description**	**Reported Malware**
1	VMA technique	Checking telephony services	Android.os.TelephonyManager class is used to determine telephony services such as device ID, IMEI, phone number, etc.	Andr/RuSmsAT, Android.hehe
		Checking build info	Multiple malware families determine the build information to check whether execute on an emulator, e.g.,Build.MODEL.contains("Emulator").	Pincer
		Checking system properties	The malware checks the system properties such as hardware, sensors, brand name, model to determine whether its real device or a virtual machine environment.	Obad, Pincer, DenDroid
		Checking emulator related files	Malware checks if QEMU or other emulated file exists.	Andr/Pornclk variant
		Time bomb	Malware after successful installation waits for a specific time to get activated.	BrainTest

Table 2. *Cont.*

S.No	Anti-Emulation Approaches	Technique	Description	Reported Malware
2	PID	Fuzzing-based Exploration	Malware checks if input data is fuzzed, e.g., read/-write files, fields of user input , exported APIs, and communication network	Drebin and Ma-MaDroid
		Model-based Exploration	It aims at injecting events aligning with a certain pattern derived by examine the app's code.	DroidBot
		Programmed Interaction	It uses the programmed interactions to evade automated runtime analysis.	Diao et al. [39]
3	CSBD	Granularity to translate and execute apps code	Apps code running in the emulated environment (QEMU), context switching does not occur, but such behavior is not seen in the actual CPU environment.	VECG [40] and Jang et al. [41]
		Process an external interrupt	App code runs in the emulated environment (QEMU) and never handles an external interrupt, while actual CPU environments support both kinds of Interrupt.	MALT [42]
4	TCBD	Cache coherence	Generate a considerable timing difference for self-modifying code that overwrites itself as it runs due to an extra layer of the caching mechanism.	EmuID [43] and Jang et al. [41]
5	UVBD	Access cache with byte granularity	The problem with memory access alignment results from earlier (and contemporary) CPUs' inability to access the cache with byte-level granularity.	UltraSPARC [44] and Jang et al. [41]

VMA: Virtual machine aware; **PID**: Programmed interaction detection; **CSBD**: Context switch-based detection; **TCBD**: Translation cache-based detection; **UVBD**: Unaligned vectorization-based detection.

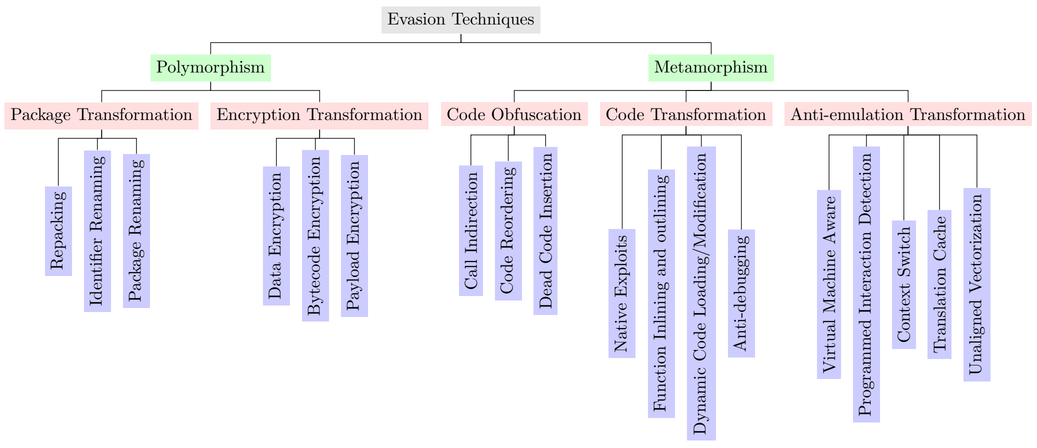

Figure 1. Taxonomy of evasion techniques for Android malware detection

Exclusion Criteria: We cannot choose all publications that match the inclusion criteria and RDs. As a result, we established specific exclusion criteria:

1. Exclude research articles not written for Android platforms such as iOS, Windows Mobile, BlackBerry, and Symbian to prevent border issues. However, these articles are applicable across all smartphone platforms, especially Android.
2. Remove papers focused on approaches that help minimize security risks rather than assessment approaches linked to malware evasion techniques. However, all the publications included that offer both prevention and detection methods of malware evasion techniques are kept.
3. Remove analysis methodologies that fail at a different level of a security evaluation, such as specific algorithms that work on app code but not on opcodes.
4. Exclude articles solely on prevention and detection methods of malware evasion techniques at the programming language level, such as C, Java (for Android), Objective-C, and Swift (for iOS), based on the dynamic, static, and hybrid framework. However, research papers that discuss security analysis methodologies in general, regardless of programming language are included.

5. Omit several research publications that focused on a specific type of attack rather than describing the prevention and detection methods of malware evasion techniques.

The proposed survey evaluates evasion techniques via a comprehensive review of Android malware detection frameworks. Unlike previous studies [17,45,46], this survey focuses on mobile malware evasion techniques. Furthermore, this investigation identified author contributions from top companies, such as S&P, IEEE Transactions on Mobile Computing, Elsevier Computers and Security, Digital Investigation, IEEE TIFS, Elsevier Future Generation Computer Systems, and ACM Computing Surveys. We present the following key contributions in light of the facts mentioned earlier.

- The proposed survey presents an evasion techniques taxonomy for the Android platform. The taxonomy systematizes and illustrates popular evasion tactics in the attacker community, their influence on novel malware that evades anti-malware, and malware evasion's impact on the analytical techniques.
- While much of the prior work has focused on the commercial anti-malware comparison, we examine academic and commercial frameworks for Android. The following study reveals the most recent Android malware analyses and challenges that restrict the identification of evasion tactics, their impact on anti-malware tools, and detection accuracy. The proposed study thoroughly investigates evasion techniques, their impact on anti-malware research, and solutions to detect persistent threats.
- The proposed survey identifies the malware evasion techniques and their detection method research gaps via a thorough comparison of various studies and frameworks through SLR (SLR is short for systematic literature review). As a result, we identified research gaps, allowing for the introduction of a comprehensive list of recommendations and a sizeable number of suggestions for future research directions.
- Finally, the survey not only tries to cover all possible advanced methods of polymorphism, metamorphism, and code transformation techniques but also provides a comparative explanation of the possible solutions or frameworks that occurred over one decade (2012–2022) by multiple tables and pie charts, which is essential to understand the important objective of our study and motivate new researchers to provide a robust future research direction.

The remaining sections of the paper are arranged in the subsequent order: Section 2 offers the required context for this research; evasion techniques and their detection on the Android smartphone platform. Section 3 covers smartphone malware identification techniques. The taxonomy of evasion techniques is presented in Section 4. Section 5 explores detection and assesses the present state-of-the-art in evasion techniques, test-bed tools, and detection frameworks. Sections 6 and 7 address the knowledge gained and future research directions. In Section 8, the article concludes the paper with important directions for further research.

2. Background

Android app components are presented in Section 2.1, and their weaknesses in Section 2.2. We highlight the importance of a few weaknesses to support this survey's necessity and clarify critical terms for the readers' benefit. Additionally, we have incorporated Android security vulnerabilities, existing obfuscation techniques, and security challenges in Sections 2.3 and 2.4. Section 2.5 briefly explains the limitations of existing anti-malware solutions.

2.1. Android Application

The phrases an *application*, *APK* file or an *app* are interchangeably used throughout this article, which refers to the Android app. Figure 2 illustrates an archive, where the unzip tool extracts the source code and associated permissions, images, and other files and directories from the compressed *APK*. We clarify the necessity of APK components and explain some critical terminology in this section. Before Android version 4.4 *KitKat*, the

Dalvik Virtual Machine (DVM) served as the virtualized environment in which the apps are executed. The later app versions use the Android Run-Time (ART); both the execution environments are analogous to the Java Virtual Machine (JVM). Various files and directories are compressed, which forms a *.apk* file. *Classes.dex* is the primary source file in-housing the executable bytecode of the Java classes declared in the source code. *AndroidManifest.xml* is a specification file containing sensitive device use permission, android component definitions, and implicit and explicit intent details. The Res folder contains uncompiled resources, while Resources '.arsc' has compiled resources like images and implicit components. The user must be installed the Android app to use the app services. Android accepts the APKs with digital certificate, also known as a developer identification. Neither a Central Authority (CA) is available to keep the records of all these developers' keys, nor is a trust chain between developers and app stores [47]. Hence, the Google Android project enables a trust chain as default.

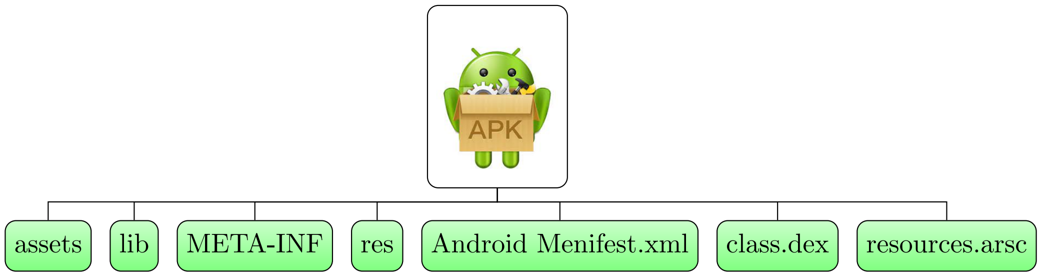

Figure 2. Android application.

The Android framework executes Android apps in the Android runtime virtual machine (ARTVM). The user apps have limited access to system resources based on the granted permissions, whereas service apps operate in the background [48]. Following are the four key components of a typical Android app:

(a) Activities: The interface with which end-users engage and use intentions to connect with several additional activities.
(b) Services: The backend component that executes the app in the background.
(c) Content providers: Content providers are intermediary elements that allow apps to share data.
(d) Broadcast and receiver intents: Distribute messages to all apps or particular apps using intent broadcast and receivers.

2.2. *App Vulnerabilities*

Adversarial malware developers identify app vulnerabilities and misuse them. Further, they exploit archived app components via:

- Device information: Malicious apps compromise app security via device identifiers such as IMEI, MEID, ESN, or IMSI. The same can be achieved by READ_PHONE_STATE permission.
- Personal information: Apps can access user contact lists, phone numbers, and calendars that can be compromised through access to contact and message permissions.
- Device/user location: Apps can approximate the user's location using the WiFi network or the network tower. Hence, location preference is a choice rather than a compulsion.
- Task monitoring: Apps record mobile and WiFi data usage data that can potentially compromise user privacy and security [49]. The GET_TASKS permission is responsible for task monitoring.
- User phone and messages: Some apps can access a user's SMS/MMS, and disrupt calls and messages, thus causing distress and misuse of information.

- User account details: Some applications can access the user's various accounts, thus jeopardizing security and putting sensitive data at risk. The GET_ACCOUNTS permission is responsible for this.
- Manipulating device storage: Apps can duplicate or format the data inside the device storage [50,51].
- Device hardware controls: Without the user's knowledge/consent, apps can record audio/video and take pictures. Some applications that ask for permission can track the mobile network traffic and monitor the background processes [52].

2.3. Android Security Vulnerabilities and Existing Obfuscation Techniques

Android apps are developed in semantically feature-rich Java language; adversaries such as malware authors, software plagiarists, and cybercriminals misuse reverse engineering tools to modify apps [53]. Software companies deploy obfuscation techniques to protect their propriety as the first layer of defense against plagiarism or cyber-attacks [37]. The authors thoroughly investigate the prevalence of obfuscation techniques used by adversaries, evasion methods that hide the app's real intent, and some practical app security challenges. During the development of this survey, we analyzed over 20,000 Google Play apps with more than ten thousand downloads. Surprisingly, less than a quarter of benign apps use obfuscation or app protection tools to save their software from cyber criminals. In contrast, cybercriminals use these encryptions, protection, evasion, and obfuscation techniques to hide their behavior and circumvent anti-malware.

Android IDE supports the basic version of ProGuard [54], thus assisting the developer in defending their proprietary code. Further, robust solutions such as DexGuard [55] are professional. However, ProGuard incurs additional costs. Since Android code is written in Java, obfuscation techniques are inherited from the parent platform. Basic obfuscation includes replacing the meaningful package name, class name, method name, or field(s) in a method with an unrelated character from a set of alphabets [a-zA-Z]. It is also possible to use 'NULL,' 'AUX,' or developer-defined strings such as 'ZERO'. Though Java-based obfuscation can be deployed in the Android SDK, the situation of deploying them is different. For example, components of Android such as activity, broadcast receiver, service, or content provider must remain un-obfuscated for smooth communication with framework API and callbacks. The same applies to a method invoked via reflection API in the Android framework. At the same time, we have covered a variety of code-transformation approaches. Setting Android-based obfuscation tools involves more than selecting features [56,57]. In addition, many complicated circumstances make obfuscating specific parts of code complex or impossible to understand, and even if that code is obscured, the app will no longer operate.

2.4. Android Security Challenges

VirusTotal reports an exponential rise in app submission at its analysis portal [58]. Adversarial malware developers employ stealth techniques, such as code obfuscation [59], dynamic code loading [60], encryption, and repackaging [61], to dodge commercial anti-malware [62] including Bouncer, the Google play anti-malware. The persistent malware accomplishes its goals by employing similar techniques, deceiving the detectors based on the signature. Improved techniques match the mobile platform and provide a rapid response for the Android OS. A smartphone can be controlled by exposing the app or OS vulnerabilities and obtaining sensitive data [61], receiving monetary benefits by adversely utilizing telephone services, or forming a botnet. In the following, we present some issues with Android malware investigation and identification:

(a) Automated signature generation: Transformation approaches and code obfuscation techniques are more common in signature-based recognition procedures. However, these methods must also regularly keep their databases up to date by adding malware variants with minor changes. Furthermore, the signatures are manually analyzed and extracted, which takes time and skill, while creating signatures for multiple threats may generate false negatives. The significant increase in malware variants necessitates the automated generation of malware signatures. It will assist in reducing the number of misleading malware detection alerts. Offline analysis approaches are required to comprehend the fundamentals of malware operation.

(b) Smart devices resource constraints: Battery-powered smartphones are constrained due to the processing speed, memory, battery power, or limited storage capabilities. Hence, anti-malware techniques suitable for computing devices are unsuitable for devices with limited battery; thus, there is a need to identify cloud-based analysis and detection techniques [63,64].

(c) App collusion: Recent studies evaluate a single app, overlooking the impact of collusion between two or more app attacks. As a result, there is a need for a paradigm shift in the assessment of smart devices that evaluates the interaction of numerous apps [65]. Compositional vulnerabilities are revealed by pushing from analyzing one app to multiple (colluding apps) analyses at the system level. The malware developers may exploit bugs in various benign apps. Furthermore, much current research concentrates on a particular application or system element [66]. The adversaries develop fragmented malicious functionality among two or more apps where anti-malware declares a single app genuine.

(d) Native code: Native and dynamic loading codes can control the hardware due to their direct execution capabilities, which remains a significant challenge for Android security [65].

(e) Tools availability: Around 20% of the publications have publicly provided their research artifacts. To stimulate the research community, researchers could provide innovative tools and develop libraries [65].

(f) Tools applicability: With resource constraints, the early conventional security mechanisms are no longer relevant as they require more computation power and resources to be implemented [66,67].

(g) Ransomware attacks: With Android malware evolving, the trend has shifted from stealing data to hijacking devices, encrypting data, and demanding a ransom in exchange for punitive damages to a user. Since resetting a device is no longer a solution against ransomware attacks, robust detection and prevention systems must be developed to prevent further compromise of data and security [68].

(h) Scalability: Scalability is still an issue when examining many apps. Hence, a scalable system [69] to identify malware and combat malware attacks is important [69].

(i) Reactive static, dynamic analysis techniques are evaded by persistent attacks [70]. Furthermore, advanced malware employs anti-emulation methods to avoid dynamic analysis [71,72].

2.5. Anti-Malware Issues

Simple modifications can easily circumvent signature-based anti-malware solutions. Dynamic code execution facilities are exploited to evade dynamic systems such as Google Bouncer [73]. The core functions of Google Bouncer and related anti-malware techniques and technologies are not public. However, researchers have demonstrated ways to analyze security systems via different methodologies. For example, DroidChameleon [21] proposed a framework that can automatically submit obfuscated apps to ten prominent antivirus solutions, bypassing the top commercial anti-malware. More than 86% of reported malware use repackaging [74] techniques to bypass anti-malware. Therefore, it is critical when around 43% of malicious app signatures do not rely on code-level artifacts and can be removed with a few simple APK or manifest modifications.

3. Mobile Malware

Malware invades a smartphone with malicious software without the user's consent. Malware is developed by combining malicious code and an application. Malware is a lethal stealth weapon for any active cyberattack. As more organizations attempt to address this issue, web-based malware distribution has increased alarmingly. Figure 3 illustrates an overview of mobile malware.

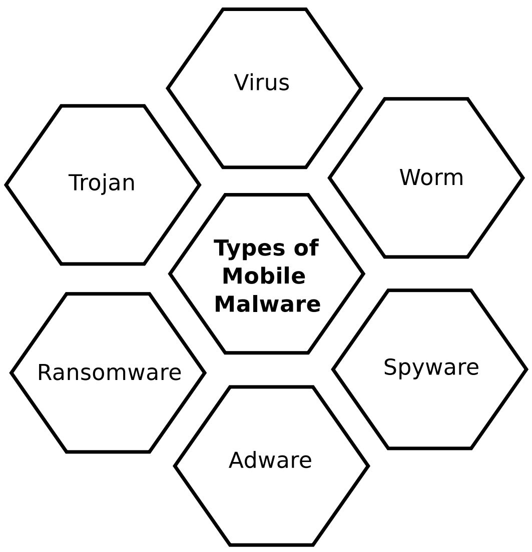

Figure 3. Types of mobile malware.

3.1. Malware Propagation

Table 3 lists the common techniques deployed by attackers to spread malicious apps. The goal is to delay or resist malware detection and analysis capabilities.

Table 3. Prominent propagation methods.

Technique	Description	Reported Malware
Repackaging an Application	Malware authors injected the malicious code/snippet into the disassembled app download from popular app stores. It was reassembled and repackaged with a jar-signer to distribute the app in a lesser monitored store.	Pokemon Go [75]
Drive by Download	Malware developers injected malicious code onto the smartphone using adversarial URL generation, social engineering, and adverts enticing users to disclose confidential information.	Android/Not Compatible [21]
Dynamic Payload	Embedded malicious code into an application. For example, using a file or APK to hide the malicious payload. By distributing a bogus vital update to an existing app, the virus is launched when the unknowing user clicks "Yes," thus compromising the smartphone.	BaseBridge [38], Anserverbot [76], DroidKungFuUpdate [77]
Stealth Malware Techniques	Malware developers employed stealth techniques, such as dynamic code loading, obfuscation, reflection, and native code execution, to avoid detection by anti-malware tools. For example, code obfuscation seeks to encrypt/unread the source code, making virus identification more difficult.	DroidDream [78], Android.Opfake

3.2. Malware Behaviour

Malware writers use intelligent and advanced techniques for their activation and installation methods. Most importantly, malware writers take good care of its persistence in the device and its evasion methods. This work thoroughly analyses a few select malware families' behaviors and presents them in Table 4.

Table 4. Prominent Malware Behaviour.

Malware	**Activation**			**Persistence**		**Anti-Analysis**				
	Event	**Host**	**Scheduled**	**Stealthy**	**Prevent Destroy**	**Renaming**	**String Encryption**	**Dynamic Loading**	**Native Payload**	**Evade Dynamic Analysis**
Airpush		Y	Y			Y				
AndroRAT	Y		Y							
Aples	Y		Y		LD					
BankBot	Y		Y	BL, HI	MDA	Y	Y	Y		CDI
Bankun	Y		Y	BL, HI						CDI
Boxer	Y					Y	Y			
Dowgin	Y	Y				Y	Y	Y		EC
DroidKungFu	Y	Y	Y		KA	Y	Y		Y	
FakeAV	Y		Y	BL						
FakeAngry	Y					Y	Y			
FakePlayer	Y			BL		Y				
FakeTimer	Y		Y							
FakeUpdates	Y	Y	Y			Y	Y			
Fobus [2014]	Y			BL, HI			Y	Y		EC
GingerMaster	Y	Y	Y			Y	Y			
GoldDream	Y									
Gorpo						Y	Y			EC
Yisut	Y				LD					
Kemoge	Y									
Koler	Y				LD, MDA	Y				CDI
Ksapp	Y	Y	Y							EC
Kuguo	Y	Y				Y				
Leech	Y	Y	Y	BL	MDA	Y	Y	Y		EC
Lotoor	Y					Y		Y	Y	
Obad	Y		Y	BL, HI	HA	Y	Y			CDI, EC
Opfake	Y		Y	BL, CL, HI			Y			
RuMMS	Y		Y	BL, HI		Y	Y	Y		
SimpleLocker	Y		Y	HI	LD	Y				
SlemBunk	Y		Y	BL, HI			Y	Y	Y	
SmsZombie	Y		Y	BL, CL						
SpyBubble	Y		Y	BL, CL						
Svpeng	Y			BL	LD					CDI
Triada	Y		Y	CL, RK	SYS	Y	Y	Y		CDI, CIA
UpdtKiller	Y	Y	Y	BL	KA, MDA	Y			Y	
VikingHorde	Y		Y		RI				Y	
Vmvol	Y			BL, CL						
Winge	Y					Y				
Youmi		Y				Y				
Zitmo	Y		Y	BL, HI		Y				
Ztorg		Y				Y	Y	Y		EC

Stealthy: Block (BL); Clean (CL); Hide Icon (HI); Rootkit (RK). **Prevent destroy**: Hide Admin (HA); Kill AV (KA); Lock Device (LD); Monitor Destroy Action (MDA); Reinstall (RI); System App (SYS). **Evade Dynamic Analysis**: Check Device Info (CDI); Encrypt Communication (EC); Check Installed App (CIA).

3.2.1. Activation

The examination of Android malware reveals three activation methods: host-app activation, scheduling, and event-based activation approaches. The integrated repackaging approach, in which the adversary inserts malicious code into the apps to activate the malware with the hosted app, is analogous to the through-host-app activation technique. The scheduling option is also commonly used to track or collect information regularly. They usually employ Android's AlarmManager with pending intent or register a thread of timer tasks.Ransomware makes extensive utilization of scheduling. Some ransomware applications work a recurring task at a very short interval, rendering the victim's device unresponsive.

3.2.2. Persistence

One of the essential characteristics malware authors consider when developing an app is persistence. Further, the longer the malware remains in the victim's device, the more substantial sales the adversary may be able to generate. Nevertheless, persistence may be deployed in several ways, such as:

(a) Keeping the malware's existence as undetectable as possible. We discovered many stealthy approaches used by malware to hide traces of malicious threats:

  (i) Objects such as calls, SMS, notifications, and music can be blocked.
  (ii) Cleaning gadgets of devices such as SMS history and name logs are essential for the infection since automatically dispatched messages or contact information may potentially warn the victim that something is amiss.
  (iii) Even though the background carrier is operating, the malware launcher symbol is hidden [79–82].
  (iv) To conceal its existence, it exploits gadget APIs.

(b) Using strategies such as masking itself from displaying in the list of device administrators, locking the device, disabling the antivirus process, and so on, prevent itself from being destroyed by the device, antivirus software, or a human.

3.2.3. Anti-Analysis Techniques

(a) One behavior pattern in each approachis renaming, followed by obfuscation tactics. First, the important name of the argument, function/methods, classes, and package are renamed into relatively incomprehensible or meaningless forms. This transition makes manual assessment much more difficult. On the other hand, renaming does not affect API calls and static evaluation techniques [83].

(b) String encryption is also commonly found in malicious apps. Researchers and anti-virus software can identify malware by looking for strings in the source code, such as the fundamental values of JSON/XML, URL of server URL, rationale action, reflection, and method invoke strings. Moreover, malware may employ string encryption to exchange plaintext for ciphertext, making it more challenging to analyze malware behavior. The malware frequently uses some or all of the given string encryption techniques: one-time pad, byte permutation, DES/AES, and base64 encoding. In order to analyze malware manually, the decryption and decoding process must be repeated and traced back to a simple textual form [84].

(c) Dynamic loading: The .dex file has recently become increasingly popular. It generally comprises a small dropper payload that seems benign. However, the resource directory (such as RuMMS) or valuable assets are loaded into the actual payload from dropper payloads or downloaded from the web (e.g., SlemBunk). The actual payload is encrypted to complicate the assessment process further (e.g., Fobus).

(d) Native payload: As a result, the native library is a perfect place to hide malicious activities. Native payloads are becoming increasingly popular, according to our research. Malicious apps hide features and sensitive data within the local code, such as top-class numbers and the server URL.

(e) Evade Dynamic Analysis: The primary notion behind escaping dynamic analysis is to identify the current execution environment of the malware, e.g., activate BankBot [85]; this checks the model number, IMEI number, device manufacturer, brand, and specific fingerprint value. If the executing environment fulfills the criterion, the malware will operate benignly, prevent itself, and stay hidden. Another complex spyware, Triada, will check whether the IMEI meets particular patterns and if "com.Qihoo. Androidsandbox" is pre-installed and running; it also behaves benignly. Some malware encrypts the connection with its command and control (C&C) servers to avoid dynamic analysis that carefully examines the malware's communication channel [85–87].

4. Evasion Techniques

Many approaches and technologies have been made accessible, and Android app developers have been using them to safeguard their intellectual property. However, it is worth noting that methods and procedures created to safeguard intellectual property are routinely exploited and misused by adversaries to deploy stealth malware apps. Malware writers use evasion techniques to dodge detection and float the malicious modules among genuine apps after they have been identified in the wild. The research community has witnessed evasion techniques across many Android malware families. The evasion techniques are rapidly evolving with new features, such as sensing an app's execution environment (sensing an emulator or real device to hide the malicious functionality of the app under observation) to defeat anti-malware apps [12]. Furthermore, malware writers can use an appropriate evasion technique to change the entire app structure and harden reverse engineering. Hence, this has become an obstacle for security researchers to understand the malware functionality.

Since smartphones are battery-constrained, on-device resource-extensive anti-malware apps are unsuitable [88]. The adversary exploits the misuse of these constraints via code obfuscation to drain the smartphone's battery. Hence, developing suitable evasion techniques can be classified as polymorphic and metamorphic.

Figure 1 illustrates a detailed taxonomy of evasion techniques discussed in the subsequent sections.

4.1. Polymorphism

Polymorphism is a code evasion technique. A stealth mechanism that uses obfuscation and encryption to transform malicious apps into different forms (i.e., change the appearance while keeping their functionality intact). Polymorphism can be further classified into package and encryption transformation.

4.1.1. Package Transformation

Package transformation concerning modifying the code such that the anti-malware identifies the modified sample as unseen malware. Repacking (RPK), identifier renaming (IDR), and package renaming (PKR) are three methods of package transformation.

(a) Repacking (RPK): A popular downloadable app from the official or third-party market. Then, it is disassembled using reverse engineering tools, apktool. The malicious code or payload is subsequently inserted into the benign app and reassembled. Finally, malware developers use custom keys with jarsigner and release them at local app stores. The malicious code is encrypted in a '.dex' file. The Malgenome dataset contains more than 80% of repackaged malware variants of apps available at legitimate third-party stores [89–91]. Glanz et al. [92] reported 15% repacked apps. Repacking enables unseen samples, causing the anti-malware to fail to detect them [21,37,93]. Pokemon Go [75] and Anserverbot [76] samples are known repackaged malware.

(b) Identifier renaming (IDR): Dalvik bytecode binds the identifiers such as classes, methods, and fields. The names of the identifiers are modified, keeping the code semantics intact [94,95]. The words are replaced by meaningless or puzzling strings (e.g., llllooooooo). However, the constructors and methods override the superclass and cannot be renamed. The identifiers are replaced by either sequentially generated string laterals, such as *a*, *b*, *c*, *d*, or progressive numbers, such as 1, 2, 3, 4 [96]. Figure 3 illustrates the renaming of the identifiers of a class called Sum. Plankton, Geinimi, and BaseBridge malware use the identifier renaming (as shown in Listing 1) evasion technique.

(c) Package renaming (PKR): The Android ecosystem identifies each app with a unique package name. In this technique, malware writers rename the app package name in the AndroidManifest.xml. Some anti-malware identifies the malicious app signatures with simple parameter values, the name of the method, and its class and imported packages. However, such a vague technique helps adversaries quickly change the malicious sample signature [97,98].

Listing 1. Identifier Renaming.

```
public class a {
   private Integer a;
    private Float = b;
       public void a (Integer a, Float b){
       this.a=a+Integer.valueOf(b)}
    }
}
```

4.1.2. Encryption Transformation

The encryption techniques are preferred over data, bytecode, or malware payload. Payload encryption (PEN), data encryption (DEN), and bytecode encryption (BEN) are three encryption types.

(a) Data encryption (DEN): This is more complicated when compared with identifier renaming. Dalvik files store data such as strings and arrays in their data structures. It is noted that string-like messages, network addresses, and shell commands reveal app information. In an obfuscated app, strings or plain text can be encrypted by random puzzling strings and harden reverse engineering [99]. These strings can only be decrypted at the runtime evading static analysis [100]. The DroidDream [78] and Bgserv [101] malware families extensively employ data encryption.

(b) Bytecode encryption (BEN): Bytecode encryption aims to encrypt the bytecode to bypass static analysis. The malicious code is encrypted using this technique and can only be decrypted via a decrypt routine at runtime. In this way, the decryption routine remains available to signature-based methods. Every malicious code variant possesses a decrypt routine (obfuscated in different ways).

(c) Payload encryption (PEN): Malware writers let malicious applications carry suspicious encrypted payloads. These additional payloads are installed onto the user's device once the system is compromised. Malware such as DroidDream [78] exhibits this kind of behavior.

4.2. Metamorphism

Metamorphic malware leverages obfuscation techniques at its best to evolve its body to produce new variants. The code is mutated and no longer looks the same but possesses the same behavior. Metamorphism techniques do not comprise the encryption part, unlike polymorphism techniques. Instead, metamorphism techniques employ a mutation engine to mutate its own body. Every variant has a different code size, structure, and sequence, resulting in a well-constructed metamorphic variant while preserving the original program logic and behavior. Metamorphism can be categorized into code obfuscation, code transformation, and anti-emulation transformation.

4.2.1. Code Obfuscation

Code obfuscation or mutation techniques hinder anti-malware detection systems via code changes from one generation to another. Third-party developers use obfuscation or sensitive algorithms to protect their intellectual property from plagiarism [102]. In contrast, cybercriminals employ code obfuscation to protect their malicious behavior and avoid anti-malware detection. Call indirection (CIN), code reordering (CRE), and dead code insertion (DCI) are three types of code obfuscation techniques [103].

(a) Call indirection (CIN): The call graph defines the caller–callee relationship between the different app modules. The call graph creates a semantic signature of a given app. Call indirection aims to manipulate these call graphs (calling non-existing methods previously) and prevent or delay detection. This obfuscation can be achieved for all calls, whether calls are made within the app code or in the framework library. The foremost aim is to defeat automatic anti-malware analysis.

(b) Code reordering (CRE): Code reordering aims to reorder the set of instructions in a program [104]. CRE employs 'goto' statements to preserve the order of instructions during execution. The code reordering technique can alter the random code instruction signature reordering and evade detection tools.

(c) Dead code insertion (DCI): DCI transformation injects dead or irrelevant code blocks into the program. It is intended to increase the app size with its analysis time by keeping the original app's semantics (i.e., it does not affect the rest of the code). In addition, dead code insertion modifies the opcode order, changing the signature of malicious apps. DCI defeats opcode-based detection systems and signature-based anti-malware.

4.2.2. Code Transformation

Code transformation techniques hinder disassembly tools [105] and evade commercial anti-malware [37,106,107]. These techniques obfuscate existing malware samples to generate unseen malicious files. Native exploits (NEX), reflection API (REF), function inlining and outlining (FIO), anti-debugging (ADE), and dynamic code loading (DCM/DCL) are code transformation types.

(a) Native exploits (NEX): App archives combine Java source code, native libraries, manifest declarations, and resources. The native code uses C or C++ for performance improvement and portability. Adversaries misuse the native code to hide malicious behavior. Moreover, they encrypt native code and hide them in non-standardized places. Therefore, the detection system designed for non-native applications may not work for native applications.
A packer can convert identified functions into native methods of a '.dex' file, which can be loaded using the JNI (Java native interface) methods `dvmLoadNativeCode()` [14]. Android allows users to directly execute native code and machine code on a smartphone processor. The JNI, a predefined interface for communicating between native and Java code, is the most widely used method to invoke native code on the Android platform. Launching native code at the root level implies that DVM or ART imports a shared object of Linux OS and allows calling the native methods stored within it. Java and native code share a common sandbox. Therefore, the same permissions are imposed on Java and native code. Native code on Android is subject to almost the same privilege constraints as Java code. For instance, an application cannot open an Internet socket from native code without permission to access the Internet. However, attackers have one distinctive advantage when running native code: While they go through a well-defined API to load code into the Java environment, they can quickly load and execute code from the native executable in various ways. The fundamental advantage for attackers is that there is no distinction between code and data at the native level. Java requires an app to load the class file on the processor; hence, an adversary can execute the native code.

(b) Function inlining and outlining (FIO): Function inlining and outlining are code optimization techniques to reduce the overhead of the call. They are also used as resilient obfuscation techniques. The inlining technique replaces a function call with the actual function code. Thus, the transformation results in a different app with the original functionality.
On the other hand, a statement set is put into a function or method in the outlining technique. Function inlining and outlining, when used together, act as sound obfuscators. Reflection is a powerful feature of the Java language that allows developers to interact with programs at runtime. It accesses the class information to create new object instances and invokes the runtime method with string literals. Nevertheless, searching for reflection API is easier once the string literals are encrypted. The reflection API hides the malicious behavior as it can implicitly transfer control to the functions. Malware developers create stealth malware via code outlining.
Without explicit Internet connection authorization, the app cannot establish an Internet socket through native code [108]. Since there is no restriction on native code execution, the attacker can execute native code on the CPU. Therefore, they must use predefined APIs to import code into the JRE. However, the native executable code can be loaded and executed immediately. In addition, at the native level, there is no separation between data and code, whereas Java needs a program to load a class file to launch its code manually. These factors significantly reduce native code protection.

(c) Dynamic code loading/modification (DCL/DCM): Android apps are written in Java but must be converted into Dalvik bytecode with the .dex tool, executed by the virtual machine. Adversaries execute native code via the JNI. Hence, they can exploit dynamic code loading at runtime to execute the exploits.
The malicious dynamic payloads can be hidden inside the app as an external .jar file [109]. Since the payload is encrypted, it can defeat static analysis techniques. Malware writers decrypt and load the malware at runtime [110–116]. For instance, the malicious code prompts the user for a critical update; hence, a novice user may consent, causing the device to become infected by the malware.
The Android framework has allowed external code loading since its first API version. Adversaries execute the code at runtime with the available app permissions. Google's content policy [117] also concedes third-party developers' runtime code from the installation package only. However, the unwanted apps and malware fetch undesirable code from remote command and control servers [117].
Qu et al. [117] studied effects such as (i) local or remote code fetching; (ii) security implications among off-the-shelf apps; and (iii) integrity verification against encrypted `.dex` loading. Poeplau et al. [14] studied 1632 popular apps with over one million downloads and reported that 10% of the apps were vulnerable to DCL injection attacks. Apps using DCL are subject to code injection at the update time. Popular code packers, Ijiami and Bangcle, load encrypted bytecode at runtime and decrypt the same in memory [117].

(d) Anti-debugging (ADE): Anti-debugging is a popular anti-analysis technique that identifies being executed in debugging mode. Hence, the attack code remains silent upon sensing the debugging environment [118–121]. Stealth malware triggers the desired malicious behavior at the correct instance. Under the unfavorable scenario, they behave genuinely, suspending the hostile actions and changing the original code to crash or altering the original execution path. The virtual malware setup fails to identify such instances, creating an overhead. Anti-debugging techniques based on the Java debug wire protocol (JDWP) or ptrace are preferred.

4.2.3. Anti-Emulation Transformation

Attackers deploy emulated environments, such as default Android emulators, QEMU, or Genymotion, to investigate the payload interaction in a virtual environment. If the sample identifies VMs, the app deactivates malicious payloads and behaves genuinely. The attack payload is executed if the actual devices are available. The difference between the environment of an emulator and an actual device can be determined based on their features [41,122–125]. The emulator detection strategies relying on hardware design and architecture are:

(a) Virtual machine aware (VMA)-based detection: The typical VMA techniques are described in Table 2. These techniques assist the attackers in identifying VMs and evading anti-malware tools and techniques [126].

(b) Programmed interaction detection (PID): Malware researchers analyze suspicious apps via random input data with the monkey tool. It generates the pseudo-random series of user events, such as touch and clicks, to cover all execution paths. As a result, advanced malware can identify the apps using tool-generated inputs. Furthermore, PID is restricted, especially when interacting with automated samples, because certain malicious apps disguise their wrongdoings using the user interface (UI).

(c) Detection based on context switches: The context-switch-based QEMU emulator exploits the race condition among two threads to identify suitable locking mechanisms. The method neither needs timing disparity nor kernel privileges. Multiple threads are concurrently executed due to the hardware and OS support for context switching [127,128]. An involuntary context switch occurs when the CPU is interrupted by an external timer event. The following facts can be seen based on context switching and the QEMU emulator interrupt handling technique; consequently, we can build a method to identify the QEMU environment accordingly. First, context switching seldom occurs in the QEMU scenario when processing a basic block, as stated in the introduction. Nevertheless, this behavior is not found in actual CPU execution scenarios. Instead, instructions within a QEMU primary block environment are processed atomically, although the device does not find atomicity. We can use this functionality to distinguish between the QEMU emulator and the actual CPU by executing a multi-threaded application with the problem of race conditions. In a typical CPU scenario, we can attain the state of the race condition by executing this code, but in a QEMU emulator environment, the race condition seldom occurs.

(d) Detection based on the TB-cache: DBI (DBI is short for dynamic binary instrumentation)-based emulators improve efficiency via translation-caching method [129–134]. Although this caching mechanism improves emulation efficiency, it also introduces a substantial time disparity when executed on a real CPU. The same can be used to identify an emulated environment [135]. We can determine the virtual environment based on the execution efficiency of the self-modifying code.

(e) Detection based on unaligned vectorization: The inability of the CPU demand to access the cache with granularity at the byte level creates memory access problems. For instance, some 64-bit CPU designs obtain a 60-bit addressing bus for memory fetching. Because of the missing four bits, one such CPU can now only read memory if the requested memory address is a factor of sixteen. When the CPU needs to read a memory address, not a factor of sixteen, it must read the memory multiple times and reassemble the required data. The feature discriminates between the native hardware and the operating system-emulated environments without relying on the kernel. The unaligned vectorization approach beats other emulator detection techniques in many ways, making it ideal for commercial app developers and suppliers that want to secure their apps against hostile reverse engineering [136,137].

(f) Virtual machine introspection (VMI): This refers to a set of strategies for reconstructing a virtual machine monitor (VMM)'s guest context [138,139]. For instance, it is comprehending the critical kernel data structures (such as task lists) and extracting meaningful information from them. Unfortunately, having a thorough awareness of the kernel's data structures in closed-source operating systems is very challenging. Dolan-Gavitt et al. [94] developed a method to resolve this issue, automatically producing introspection tools involving observing the behavior of similar tools inside the guest system and then simulating the same processing beyond the guest system.

(g) The VMI approach is intended to record OS API calls alongside its arguments and return the result to generate a malware profile [138,139]. VMI-based methods examine how a system call appears by presuming that the monitored program implements the invoking pattern given by OS's binary application interface (ABI). They can do this while minimizing their visible footprint and reconstructing the events as if the OS were explicitly instrumented. Consequently, evasion techniques are not resistant to VMI-based methods.

5. Evaluation of Evasion Detection Frameworks

The authors looked at studies published in the last decade that compared malware detection frameworks as opposed to the evasion tactics covered in Section 4. From 2011 to 2021, the authors examined Android malware platforms and the robustness of these frameworks against listed polymorphism evasion techniques, as listed in Table 5 in Appendix 1. The indications "√", "×", or an empty cell that intersects the framework row with the evasion column identify researchers who tested their framework against certain evasions [140]. "√" indicates that the study either tested or assumed it could detect the evasion tactic. At the same time, "×" means the researcher assumed that the evasion technique bypassed their Android malware detection framework. Incomplete reports of framework evaluation studies on evasion tactics or assumptions are shown by an empty cell. For example, DroidMat [141,142], MAMA [143], QuantDroid [144], DenDroid [145], Sheen et al. [146,147], and RAPID [148] insufficiently reported the evaluation of their proposed detection framework against evasion techniques.

Table 5. Evaluation of static, dynamic and hybrid frameworks against polymorphism evasion techniques.

		Polymorphism Evasion Techniques								
Framework	**Year**	**Package Transformation**			**Encryption**			**Analysis Type**		
		RPK	**PKR**	**IDR**	**DEN**	**BEN**	**PEN**	**ST**	**DY**	**HY**
DroidMOSS [142]	2012	√						√		
RiskRanker [149]	2012				√	√	√			√
MobSafe [150]	2012									√
DroidOLytics [151]	2013	√	√	√				√		
DroidAPIMiner [61]	2013					√		√		
QuantDroid [144]	2013							√		
Amos et al. [152]	2013								√	
AndroTotal [153]	2013								√	
Shalaginov et al. [154]	2013									√
ARIGUMA [155]	2013									√
Petsas et al. [115]	2013									√
ViewDroid [156]	2014	√						√		
DroidGraph [63]	2014	√						√		
MysteryChecker [157]	2014	√			√	√	√	√		
ResDroid [158]	2014				×	×	×	√		
Lee & Kim et al. [124]	2014								√	

Table 5. *Cont.*

Polymorphism Evasion Techniques										
Framework	**Year**	**Package Transformation**			**Encryption**			**Analysis Type**		
		RPK	**PKR**	**IDR**	**DEN**	**BEN**	**PEN**	**ST**	**DY**	**HY**
TaintDroid [159]	2014								√	
Pektas et al. [160]	2014								√	
Soh et al. [161]	2014	√	√	√					√	
Shabtai et al. [162]	2014								√	
VetDroid [163]	2014								√	
DroidBarrier [164]	2014								√	
Droid-Sec [165]	2014									√
AMDetector [166]	2014									√
Chen et al. [15]	2015	√	√	√				√		
APK Auditor [147]	2015	×	×	×	×	×	×	√		
Dempster–Shafe [167]	2015	√						√		
Dexhunter [70]	2015				√	√	√	√		
DroidExec [168]	2015	√						√		
AnDarwin [64]	2015	√	√					√		
AndroSimilar [169]	2015	√	√	√	√			√		
ngrams [170]	2015	√	√	√				√		
SeqMalSpec [18]	2015			√				√		
DroidEagle [171]	2015	√						√		
Sheen et al. [146]	2015							√		
Droidkin [172]	2015	√	√	√	√	√	√	√		
Shen et al. [173]	2015	√			√			√		
SherlockDroid [174]	2015				√	√	√	√		
Kuhnel et al. [175]	2015			√	√	√	√	√		
APSET [176]	2015								√	
Afonso et al. [177]	2015								√	
Maier et al. [178]	2015								√	
Singh et al. [179]	2015								√	
Gheorghe et al. [180]	2015								√	
DWroidDump [181]	2015				√	√	√		√	
Ng, D. V et al. [182]	2015								√	
GroddDroid [183]	2015								√	
Wu et al. [184]	2015	√							√	
MARVIN [185]	2015									√
Mobile-Sandbox [71]	2015				√	√	√			√
StaDynA [69]	2015									√
Tap-Wave-Rub [186]	2015									√
Gurulian et al. [187]	2016	√	√	√				√		
TriggerScope [188]	2016							√		

Table 5. *Cont.*

Polymorphism Evasion Techniques										
Framework	**Year**	**Package Transformation**			**Encryption**			**Analysis Type**		
		RPK	**PKR**	**IDR**	**DEN**	**BEN**	**PEN**	**ST**	**DY**	**HY**
Wu et al. 2016 [184]	2016	×						√		
AAMO [34]	2016	√	√	√	√	√	√	√		
Wang et al. [68]	2016					√		√		
MocDroid [189]	2016							√		
Battista et al. [190]	2016		√	√				√		
RAPID [148]	2016							√		
DynaLog [191]	2016								√	
Q-Floid [192]	2016								√	
Diao et al. [39]	2016								√	
Droiddetector [193]	2016									√
Andro-Dumpsys [194]	2016									√
Droidsieve [195]	2017				√	√	√	√		
Ordol et al. [196]	2017		√	√	√	√			√	
Hydroid [197]	2017		√	√	√	√	√			√
Ares et al. [198]	2018							√		
BACCI [199]	2018	√	√	√	√			√	√	√
Droidcat [200]	2018		√	√	√				√	
AndrODet [201]	2019	√	√	√	√			√		
Dadidroid [202]	2019	√	√	√	√	√	√	√		
Obfusifier [203]	2019	√	√	√				√		
Kim et al. [204]	2019		√	√	√			√		
Amin et al. [95]	2020	√						√		
Alazab et al. [205]	2020				√			√		
DAMBA [206]	2020				√	√	√	√	√	√
IMCFN [207]	2020	√	√	√	√			√	√	√
Alrzini et al. [208]	2020				√	√	√		√	
Alrzini et al. [208]	2020				√	√	√			√
Karbab et al. [209]	2021	√		√	√		√	√		
BLADE [210]	2021		√		√			√		
Dharmalingam et al. [211]	2021		√	√		√		√		
IntDroid [212]	2021				√	√	√	√		
PetaDroid [209]	2021	√	√	√	√		√	√		
CamoDroid [6]	2022		√	√			√	√		
Molina et al. [213]	2023	√	√	√	√			√		√

RPK: Repacking, **PKR:** Package Renaming, **IDR:** Identifier Renaming **DEN:** Data Encryption, **BEN:** Bytecode Encryption, **PEN:** Payload Encryption **ST:** Static Analysis, **DY:** Dynamic Analysis, **HY:** Hybrid Analysis.

5.1. Polymorphism Evasion Detection

The authors contrast the static, dynamic, and hybrid frameworks against polymorphism evasion approaches. Static, dynamic analysis, and their combination, also known as hybrid analysis-based malware detection approaches, are shown in Table 5. In the two areas of encryption and package transformation, we analyze every framework against polymorphism transformation strategies. Each framework employs different datasets with a specific count of benign Android and malicious apps in the evaluation process. For illustration, APK Auditor [147] examined its framework with 1853 benign and 6909 malware apps, totaling 8762 apps downloaded from the Google Play Store and some other datasets such as Contagio (http://contagiomobile.deependresearch.org/index.html, accessed on 1 March 2023) and Genome Project (malgenomeproject.org, accessed on 1 March 2023). APK Auditor detected malware with an accuracy of 88%. Many evasion techniques restrict the identifying malware apps by the APK Auditor framework, even though it is signature-based.

5.1.1. Package Transformation

(a) Repacking evasion detection (RPK): Various detection approaches, including static analysis, can be used to identify repacking evasion. Dempster–Shafe [167] used a control flow graph (CFG) to explore the repacking features and claimed enhanced resilience to app obfuscation techniques. Similarly, Droidgraph [63] used the level of hierarchical class to identify which malicious code was repackaged from the original APK. This accounts for garbage code, code obfuscation, and API call requests [205,214]. Compared to the native call graph approaches with polynomial time, the code comparison time decreased. On the other hand, reflection eludes detection frameworks that use the control flow graph. Some static analysis methods, such as AnDarwin [64], AndroSimilar [169], ngrams [170], DroidEagle [171], DroidKin [172], DroidOLytics [151], MystryChecker [157], AndroSimilar and AAMO, are capable of detecting RPK evasions. In contrast, most works on dynamic analysis give minimal consideration to RPK evasion. Two dynamic analysis frameworks, developed by Wu et al. ([184]) and Soh et al. ([161]), were compared to RPK evasion methods in their research articles.

(b) Package renaming detection (PKR): Some frameworks based on static analysis, such as Droidkin [172] and DroidOLytics [151], were tested for their capacity to identify PKR evasion techniques. However, other works, including Andro-Tracer [106], APK Auditor [147], DroidGraph [63], COVERT [215] and Vulhunter [216] inadequately analyzed their approaches against PKR evasion, as summarized in Table 5. Furthermore, some studies analyzing app frameworks based on dynamic and hybrid analysis techniques are incompetent to investigate their robustness against PKR evasion, except for one research study conducted by Shen et al. [173].

(c) Evasion detection (IDR): DroidOLytics [151], AndroSimilar [169], Droidkin [172], Kuhnel [175], Triggerscope [188], AAMO [34], and Battista et al. [190] claim that their Android static framework for malware identification can identify IDR evasion, as shown in Table 5. Unfortunately, several other researchers underestimate its resistance to IDR evasion [217]. Table 5 illustrates the problem of ensuring the resilience of Android frameworks for malware identification over IDR evasion approaches and examines the research's framework concerning approaches to IDR evasion.

In conclusion, most static analysis-based Android frameworks for malware analysis systems can identify package transformation strategies (RPK, IDR, and PKR). On the other hand, most detection systems relying on dynamic and hybrid analysis do not sufficiently assess or report their resistance to IDR evasion tactics [218].

5.1.2. Encryption Transformation Evasion Detection

In conclusion, static analysis reveals evasion techniques based on encryption, demonstrated by DroidKin [172], DexHunter [70], Kuhnel [175], AAMO [34], and Sherlockdroid [174], are capable of identifying BEN, PEN, and DEN encryption evasions. MysteryChecker [157], AndroSimilar [169], DroidKin [172], AAMO [34], Shen [173], Kuhnel [175] and SSherlockDroid [174] were able to identify DEN evasions using static analysis-based detection techniques. Similarly, Q-Floid [192] and Soh [161] reported resistance to BEN evasions. Some detection techniques are based on dynamic analysis. Instead of using a de-compilation tool, DWroidDump [181] used executable code by accomplishing code extraction from the memory of DVM [219], hampered by the three encryption evasion methodologies listed in Table 5. Despite this, the RiskRanker framework [149] based on hybrid analysis techniques reported accurate identification of BEN, PEN, and DEN encryption evasion. Other hybrid analysis-based frameworks, such as Mobile-Sandbox [71], MARVIN [185], and AMDetector [166], tested their frameworks for BEN and DEN encryption evasion and claimed accurately identification. Likewise, the framework DWroidDump [181] also tested for encryption evasion strategies.

5.2. Metamorphism Evasion Detection

The Table 6 show the robustness of the framework used to identify malicious Android apps using dynamic analysis, static analysis, and their combination, also known as hybrid analysis. It also considers some advanced detection techniques for metamorphism evasion.

Table 6. Evaluation of static, dynamic and hybrid frameworks against metamorphism evasion techniques.

		Metamorphism Evasion Techniques												
Framework	**Year**	**Code Obfuscation**			**Code Transformation**					**Anti-Emulation**		**Analysis Type**		
		CRE	**CIN**	**DCI**	**NEX**	**FIO**	**REF**	**DCL**	**ADE**	**VMA**	**PID**	**ST**	**DY**	**HY**
Chaugule et al. [220]	2011										√		√	
Tao [221]	2012									√			√	
DroidScope [114]	2012									√			√	
Andromaly [222]	2012												√	
RiskRanker [149]	2012	√	√	√			√	√		√				√
MobSafe [150]	2012													√
DroidOLytics [151]	2013	√	√	√								√		
DroidAPIMiner [61]	2013				√		√	√				√		
Glodek et al. [223]	2013	×	×	×								√		
Amos et al. [152]	2013												√	
AndroTotal [153]	2013												√	
Shalaginov et al. [154]	2013													√
ARIGUMA [155]	2013			√										√
Yerima et al. [224]	2014							√				√		
DroidGraph [63]	2014	√	√	√								√		
MysteryChecker [157]	2014	√										√		
AdDetect [225]	2014	√	√	√	×							√		
ResDroid [158]	2014	√					×	×				√		
DenDroid [145]	2014											√		
Lee et al. & Kim [124]	2014												√	
TaintDroid [159]	2014												√	
Pektas et al. [160]	2014									√			√	
Soh et al. [161]	2014	√	√										√	

Table 6. *Cont.*

		Metamorphism Evasion Techniques												
Framework	**Year**	**Code Obfuscation**			**Code Transformation**					**Anti-Emulation**		**Analysis Type**		
		CRE	**CIN**	**DCI**	**NEX**	**FIO**	**REF**	**DCL**	**ADE**	**VMA**	**PID**	**ST**	**DY**	**HY**
Shabtai et al. [162]	2014												√	
VetDroid [163]	2014												√	
DroidBarrier [164]	2014				√								√	
Petsas et al. [115]	2014									√				√
Droid-Sec [165]	2014													√
AMDetector [166]	2014	√	√	√						×	×			√
Chen et al. [15]	2015											√		
APK-Auditor [147]	2015	×	×	×	×	×	×	×	×	×	×	√		
Andro-Tracer [106]	2015	×	×	×	×	×	×	×	×	×	×	√		
Dempster–Shafe [167]	2015	√	√	√								√		
Dexhunter [70]	2015	√	√	√			√	√	√			√		
DroidExec [168]	2015	√	√	√								√		
AnDarwin [64]	2015	√	√									√		
AndroSimilar [169]	2015	√	√	√								√		
ngrams [170]	2015	×	×	×	×	×	×	×	×	×	×	√		
SeqMalSpec [18]	2015	√										√		
DroidEagle [171]	2015	√	√	√								√		
VulHunter [216]	2015			×								√		
Droidkin [172]	2015											√		
Shen et al. [173]	2015	√	√	√								√		
SherlockDroid [174]	2015						√					√		
Kuhnel et al. [175]	2015						√					√		
Elish et al. [7]	2015	×	×	×			×	×				√		
APSET [176]	2015												√	
Afonso et al. [177]	2015									×			√	
Maier et al. [178]	2015						√	√		√			√	
Singh et al. [179]	2015									√	√		√	
Gheorghe et al. [180]	2015												√	
DWroidDump [181]	2015												√	
Ng, D. V et al. [182]	2015												√	
GroddDroid [183]	2015									√	√		√	
Wu et al. [184]	2015												√	
MARVIN [185]	2015	√	√	√	√				×	×	×			√
Mobile-Sandbox [71]	2015	√								×	×			√
StaDynA [69]	2015						√	√						√

Table 6. *Cont.*

		Metamorphism Evasion Techniques												
Framework	**Year**	**Code Obfuscation**			**Code Transformation**					**Anti-Emulation**		**Analysis Type**		
		CRE	**CIN**	**DCI**	**NEX**	**FIO**	**REF**	**DCL**	**ADE**	**VMA**	**PID**	**ST**	**DY**	**HY**
Tap-Wave-Rub [186]	2015									√	√			√
Gurulian et al. [187]	2016	√	√	√								√		
TriggerScope [188]	2016				×		×	×				√		
DroidRA [65]	2016						√					√		
AAMO [34]	2016	√	√	√		√	√					√		
Wang et al. [74]	2016	√	√	√								√		
MocDroid [189]	2016	√	√	√								√		
Battista et al. [190]	2016	√	√	√								√		
RAPID [148]	2016											√		
DynaLog [191]	2016												√	
Q-Floid [192]	2016	√	√						×				√	
Diao et al. [39]	2016										√		√	
Droiddetector [193]	2016									×				√
Andro-Dumpsys [194]	2016								×					√
DroidSieve [197]	2017	√	√	√			√	√				√		
Abaid et al. [10]	2017							√				√		
EnDroid [226]	2018						√	√					√	
BACCI [199]	2018	√	√	√								√	√	√
Kim et al. [204]	2019		√	√								√		
AndrODet [201]	2019		√									√		
Dadidroid [202]	2019		√									√		
Obfusifier [203]	2019	√	√	√								√		
DAMBA [206]	2020		√					√				√	√	√
IMCFN [207]	2020	√		√								√	√	√
Wu et al. [227]	2021											√		
Liu et al. [228]	2021											√		
PetaDroid [209]	2021	√	√	√								√		
BLADE [210]	2021	√										√		
S3Feature [229]	2022	√	√				√					√		
ROOTECTOR [230]	2023		√		√					√		√		

CRE: Code Reordering, **CIN:** Call Indirection, **DCI:** Dead Code Insertion, **NEX:** Native Exploits, **FIO:** Function Inlining and Outlining, **REF:** Reflection API, **DCL:** Dynamic Code Loading, **ADE:** Anti-debugging, **VMA:** Virtual Machine Aware, **PID:** Programmed Interaction Detection, **ST:** Static Analysis, **DY:** Dynamic Analysis, **HY:** Hybrid Analysis.

5.2.1. Code Obfuscation Detection

CRE, CIN, and DCI are the three types of code obfuscation; the following list explains every evasion detection scheme in detail.

(a) Code reordering evasion detection (CRE): To manage and identify CRE evasions, based on static analysis are offered by SeqMalSpec [18], AnDarwin [64], and Res-Droid [158]. Similarly, utilizing the dynamic analysis sandboxing technique, Q-

Floid [192] identified CRE evasion. Furthermore, CRE evasions are detected using hybrid analysis frameworks, such as Mobile-Sandbox [71]. Despite this, CRE eludes static analysis frameworks, such as Elish et al. [7] and ngrams [170], resulting in numerous false negatives (FN).

(b) Call indirection evasion detection (CIN): The Android malware identification frameworks based on call graphs [14,216] are effectively evaded by the CIN evasion approach. Although CIN evasion is identified by some static frameworks, such as DroidGraph [63], DexHunter [70], MocDroidMartin [189], AndroSimilar [151,169], AdDetect [225], and Amandroid [231], a few fail, such as APK Andro-Tracer [106], ngrams [170], Elish et al. [7], and Wu [184]. CIN can be identified using some detection frameworks based on dynamic analysis, such as Q-Floid [161,192], and some hybrid analysis frameworks, such as MARVIN [166,185] and RiskRanker [149]. The obfuscated function call was identified by app topological signature via graphlet sampling (ACTS) from the malware sample. The ACTS framework was developed by Tianchong et al. [232].

(c) Dead code insertion evasion detection (DCI): A code similarity-based Android malware identification framework, AnDarwin [64], reported the detection of dead code insertion. The code's similarity technique employs distance vectors; thus, AnDarwin is less resistant to transforming dead code insertions [64,233].
Changes in the distance–vector code enhances the semantic gap between the code vector and dead code insertion transformation. The Q-Floid [192] framework examines the runtime behavior of a suspicious app based on dynamic analysis and presents the qualitative data flow graph (QDFG). Q-Floid is based on desktop malware detection approaches and has been reported to be able to identify obfuscated code. The QDFG [234] identifies the transformation of code obfuscation. Although, Q-Floid fails to identify Android malware when employing monitoring services. MysteryChecker [157] provides a novel attestation technique based on software identifying repackaged malware using randomly generated encryption chains and code obfuscation. DroidOLytics [151] employs statistical similarities to identify obfuscated code and repackaged apps. It creates a signature repository with dynamic length modifications to detect code cloning [235]. AndroSimilar [169] employs signature-based identification approaches, achieving a 76% accuracy. However, it has low recognition accuracy for repackaged applications and code obfuscation.

5.2.2. Advanced Code Transformation Detection

(a) NEX evasion detection: The static analysis framework DroidAPIMiner [61] reported accuracy in identifying NEX evasion, illustrated in Table 6; similarly, the hybrid analysis framework MARVIN [185] and the dynamic analysis framework DroidBarrier [164] claim to successfully identify an NEX evasion. Additionally, several static analysis frameworks come across limitations in detecting NEX evasion tactics, such as AdDetect [225], APK Auditor [147], Andro-Tracer [106], and ngrams [170].

(b) FIO evasion detection: Anti-virus solutions are compared to functional outlining and inlining FIO evasions in AAMO [34,116]. On the other hand, dynamic and hybrid studies need to evaluate the assessment of its framework over FIO evasion sufficiently.

(c) REF evasion detection: Various frameworks based on static analysis, such as DroidAPIMiner [61], DexHunter [70], SherLockDroid [174], Kuhnel [175], DroidRA [65], and AAMO [34], evaluate the performance against REF evasion detection. Similarly, Maier et al. [178] employed dynamic analysis techniques, while RiskRanker [149], and StaDynA [69], used hybrid analysis techniques to investigate REF evasion detection methods.

(d) DCL evasion detection: DroidAPIMiner [61], Poeplau [14], Dexhunter [70], Maier et al. [178], RiskRanker [149], and StaDynA [69] are among the malware identification platforms for Android OS. Other approaches, such as AndroSimilar [169], analyze their method inadequately for reflection handling and runtime code loading approaches.

(e) ADE evasion detection: DexHunter [70] can only consider ADE evasion in the static analysis framework approach. On the other hand, Q-Floid [192] dynamic analysis was inadequate to detect ADE evasion.

5.2.3. Anti-emulation Detection

Only the PID and VMA evasion techniques are used in the anti-emulation evasions explained below.

(a) Researchers combined physical Android devices with an emulator sandbox to dynamically execute apps as a defensive measure against VMA evasion [116,236–240]. Gajrani et al. [241], Hu et al. [242]. Dietzel et al. [243] offered a false responder agent that provides misleading values to the malware regarding the execution environment. Singh [179] used the detection of user interactions and anti-emulators to enhance the resilience of identifying dynamic malware [244]. Petsas et al. [115] suggested several countermeasures for different types of evasion detection, such as anti-emulation employing IMEI alteration and precise sensor simulation. However, this countermeasure was not thoroughly evaluated. Dynalog et al. [191] enhanced the performance of Android malware dynamic analysis; however, they relied on an emulation tool vulnerable to detecting emulation evasions. Vidas et al. [245] employed an actual device A5 system to capture system logs and network traffic rather than utilizing an emulator in testing based on dynamic analysis [115,246] and masquerade emulator as a legitimate device to combat VMA evasion. Anti-emulation evasion tactics are the focus of several research works. However, at the same time, some research raises red flags showing that there are not enough test beds and malware samples available to explore anti-emulation evasion (e.g., [220,221]). Nonetheless, Maier et al. [178] investigated VWA evasion and provided a method based on comparing the APK's behavior when installed on an emulator versus a physical device.

(b) The fundamental disparity between the patterns of human interaction and key runners allowed this sort of modification to elude automated dynamic analysis [191,247]. Rather than depending on emulation approaches or outdated virtualization, Daio et al. [247] monitored the automated gesture and simulated user interactions to determine if an app was under investigation or functioning normally. Some sandboxing focused on anti-emulation evasion, occurring throughout the dynamic analysis [247,248]; the majority of countermeasures have relied on hybrid analysis-based identification frameworks.

6. Discussion

The possibilities of evasion identification based on static analysis frameworks are represented in Figure 4. This figure has been utilized to comprehend the frameworks that rely on a static assessment evaluation table. The pie graph approach depicts the proliferation of static analysis-based malware analysis frameworks. Each partition describes the percentage of static analysis-based solutions that have been proven effective against a specific evasion technique. The data flow can reveal some transformation attacks unaltered by these attacks. These transformations are detectable by static analysis [115]. Methods for transformation attacks are renaming static strings (such as the names of methods and classes), dead code insertion (DCI), code reordering (CRE), changing call directions (CIN), data encryption (DEN), and encrypting payloads (PEN). For instance, 15 and 14% of static malware detection frameworks consider CRE and CIN evasion techniques, respectively. The figure also reveals a specific type of transformation attack undetectable by static analysis. In Figure 4, evasion strategies such as programmed interaction detection (PID) and virtual machine awareness (VMA) have 0 %, showing resistance to static analysis. Static code investigation techniques identified trivial code obfuscation and evasions based on package transformation. However, significantly less research is available on anti-emulation and sophisticated transformation techniques, as shown in Figure 4a.

In the case of dynamic code loading (DCL), Pektas [160] detected anti-emulation evasion by employing a dynamic analysis technique developed to handle DCL evasion malware, reaching 92% detection accuracy. However, Evasion detection based on dynamic analysis approaches is risky and time-consuming, so most researchers avoid it. For example, the dynamic analysis procedures in Mobile-Sandbox [71,249] took approximately 18 minutes to complete the analysis, which is more accurately determined by considering the hardware specs of the server and APK size.

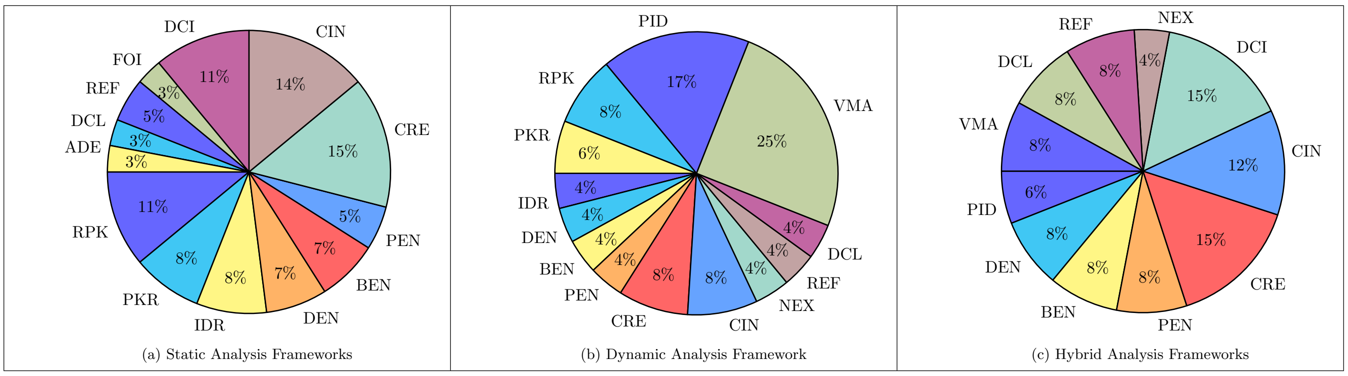

Figure 4. Malware Analysis Frameworks.

Code obfuscation can render static analysis methods worthless. The dynamic analysis technique [250] can be used to circumvent code obfuscation. As a result, sophisticated malware may be able to identify surveillance and take steps to prevent its destructive activities from being detected. Figure 4b displays the ability to detect dynamically loaded code by an app using dynamic analysis, while researchers believe dynamic analysis handles code transformation and basic obfuscation approaches. Malware samples from diverse malware families are used to test the performance accuracy of the most recent malware analysis frameworks. These samples do not reflect the proposed detection framework's true resilience against evasion strategies appropriately if the arbitrarily selected malware categories neglect evasive approaches. This is the fundamental rationale for removing accuracy from the assessment tables

The third form of evasion detection is based on hybrid analysis, necessitating a significant amount of work to gather logs and characteristics of both dynamic and static analyses. Furthermore, as demonstrated in Figure 4c, very little research has investigated their frameworks for specific evasion tactics. In 2012, RiskRanker [149] began exposing the topic of evasion and its impact on the detection accuracy rate. Anti-emulation evasions were encountered in 2014 and 2015 by Petsas [115] and Tap-Wave-Rub [186], respectively. They used the proximity sensor on the device to tell the difference between maliciously induced activities and interactions with the end user.

7. Future Research Directions

New malware variants are spreading faster than their detection and analysis, prompting efforts and improvements to build a robust malware detection framework. Based on this study, several insights were drawn and are listed below.

7.1. *Metamorphic Evasion Constraints*

After deep analysis of detection frameworks, static approaches fail to detect metamorphic evasion methods due to their dynamic characteristics. However, there is a need to build dynamic and hybrid methods to achieve success in metamorphic evasion techniques.

7.2. Standard Benchmarking

This study recommends improving the quality of Android malware literature by appending its databases with comprehensive and collaborative benchmarking frameworks. The benchmark here is a list of malware detection approaches to identify malware evasion techniques.

7.3. Android Exploits

Android is a Linux-based, open-source OS. Malware authors employ root-level vulnerabilities [251] to impact all Android versions because of the operating system's openness. An example of such an exploit is Dirty Cow CVE-20165195 [252]. Therefore, this study suggests that future researchers work on the administrative level to identify potential threats and find ways to fix all open doors for attackers.

7.4. Code Integrity Verification

Attackers use various methods to evade malware detection frameworks, with repackaging techniques by third-party authorities being one of the most popular criteria. Vidas et al. [253] developed a simple approach to solving the challenge of verifying an app's authenticity to safeguard users from malicious code in repackaged apps. Code integrity should always be maintained. Researchers and app developers should focus on maintaining the integrity of the code.

7.5. Process Authentication

Many experts utilize the model authorization approach to defend devices against various vulnerabilities without an additional certification authority (CA) [164]. However, the model authentication technique cannot locate payloads downloaded to install other malicious apps. For example, DroidBarrier [164] is meant to save such installs by identifying the underlying unauthenticated methods to thwart this attack. However, this method cannot ensure the separation of hijacked processes detailed in subsequent attacks.

As a result, monitoring tactics executed on the device are often beneficial. For example, suppose an unauthenticated process is initiated. In this case, the application should be visible to avoid causing damage to the device and analyze and track the malicious program. If malicious apps bypass all the monitoring barriers and obtain a malicious code, it will be detected when attempting to execute that code on an unauthenticated device.

7.6. Triggering Malicious Code Assurance

The system ensures that malicious code executes during dynamic analysis sandboxing. For example, TriggerScope [188] tries to find suspicious triggering by static analysis but can easily be bypassed with the help of code obfuscation techniques. Similarly, the GroddDroid framework activates all paths of each feature to verify that the malware is executed. Meanwhile, the GroddDroid [183] framework is intended to enable the execution paths of each feature to ensure the execution of malicious code. Nevertheless, it needs to recognize the branches of historical services to maintain the core activity. This is known as code coverage and is still a problem for researchers. Covering possible extensions inside the app's source code of apps is required to solve this challenge.

8. Conclusions

Android malware has become more substantial and complex due to global evasion techniques, evolving into a prominent incentive. This research exposes critical flaws in Android malware detection systems, particularly when malware employs various evasion approaches. As a result, this research looked at 88 Android malware identification frameworks and 18 assessment articles to see how successful the escape detection tactics are in Android malware identification frameworks. As a result, the study proposed a taxonomy to categorize the evasion methods into metamorphism and polymorphism. The polymorphism group includes package transformation and encryption branches, and the metamorphism group consists of three categories: anti-emulation, superior transformation, and code obfuscation.

This research also highlighted the absence of studies comparing malware identification against many prominent evasion approaches [254]. Therefore, we examine the frameworks relying upon different evasion approaches and classified reviews solely on different malware detection methods. According to the findings, few studies have examined the resilience of current Android malware identification systems to novel evasion tactics. According to this research, the detection techniques depend on static analysis, readily bypassed utilizing simple obfuscation techniques [106]. On the other hand, dynamic and hybrid approaches can deal with complex code transformation methods and cutting-edge evasion detection approaches. Nevertheless, more studies are needed to assess these frameworks regarding evasion methods.

The absence of complete test-bed tools to analyze the efficacy of existing, projected, and future frameworks need to include reviews due to a lack of significant benchmarks for evasion datasets and contemporary trending malware datasets. This work also demonstrated that detection techniques based on static and dynamic analysis should focus more on identifying evasion based on anti-emulation. Future goals include creating a uniform evaluation system that accommodates all sorts of evasion tactics and remembering a future version of malware that combines numerous evasion techniques.

Author Contributions: Conceptualization, P.F., R.B., N.E.M., and R.P.; methodology, P.F., R.B., V.J., S.B., N.E.M., and R.P.; software, P.F., R.B., V.J., S.B., and N.E.M.; validation, P.F., R.B., V.J., S.B., and N.E.M.; formal analysis, P.F., R.B., V.J., S.B., and N.E.M.; investigation, P.F., R.B., V.J., S.B., and N.E.M.; resources, P.F., R.B., V.J., S.B., and N.E.M.; data curation, P.F., R.B., V.J., S.B., and N.E.M.; writing—original draft preparation, P.F., R.B., V.J., S.B., and N.E.M.; writing—review and editing, P.F., R.B., V.J., S.B., N.E.M. and R.P.; visualization, P.F., R.B., V.J., S.B., and N.E.M.; supervision, P.F., R.B. and R.P.; project administration, P.F., R.B. and R.P.; funding acquisition, P.F., R.B., V.J., S.B., N.E.M., and R.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Vasudevan, A.; Yerraballi, R. Cobra: Fine-grained malware analysis using stealth localized-executions. In Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06), Berkeley/Oakland, CA, USA, 21–24 May 2006; pp. 15–279.
2. Egele, M.; Kruegel, C.; Kirda, E.; Yin, H.; Song, D. Dynamic spyware analysis. *Adv. Comput. Syst. Prof. Tech. Assoc.* **2007**, *18*, 1–14.
3. Palmaro, F.; Franchina, L. Beware of Unknown Areas to Notify Adversaries: Detecting Dynamic Binary Instrumentation Runtimes with Low-Level Memory Scanning. In *Intelligent Computing*; Springer: Berlin/Heidelberg, Germany, 2021; pp. 1003–1019.
4. Brumley, D.; Hartwig, C.; Liang, Z.; Newsome, J.; Song, D.; Yin, H. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*; Springer: Berlin/Heidelberg, Germany, 2008; pp. 65–88.
5. Prünster, B.; Palfinger, G.; Kollmann, C. Fides: Unleashing the Full Potential of Remote Attestation. In Proceedings of the International Conference on E-Business and Telecommunication Networks, Prague, Czech Republic, 26–26 July 2019; pp. 314–321.
6. Faghihi, F.; Zulkernine, M.; Ding, S. CamoDroid: An Android application analysis environment resilient against sandbox evasion. *J. Syst. Archit.* **2022**, *125*, 102452. [CrossRef]
7. Profiling user-trigger dependence for Android malware detection. *Comput. Secur.* **2015**, *49*, 255–273. [CrossRef]
8. Singh, J.; Singh, J. A survey on machine learning-based malware detection in executable files. *J. Syst. Archit.* **2021**, *112*, 101861. [CrossRef]
9. Singh, J.; Singh, J. Detection of malicious software by analyzing the behavioral artifacts using machine learning algorithms. *Inf. Softw. Technol.* **2020**, *121*, 106273. [CrossRef]
10. Abaid, Z.; Kaafar, M.A.; Jha, S. Quantifying the impact of adversarial evasion attacks on machine learning based android malware classifiers. In Proceedings of the 2017 IEEE 16th international symposium on network computing and applications (NCA), Cambridge, MA, USA, 30 October–1 November 2017; pp. 1–10.
11. Singh, J.; Singh, J. Assessment of supervised machine learning algorithms using dynamic API calls for malware detection. *Int. J. Comput. Appl.* **2022**, *44*, 270–277. [CrossRef]
12. Alzaylaee, M.K.; Yerima, S.Y.; Sezer, S. EMULATOR vs REAL PHONE: Android Malware Detection Using Machine Learning. In Proceedings of the IWSPA '17, 3rd ACM on International Workshop on Security And Privacy Analytics, Scottsdale, AZ, USA, 24 March 2017; pp. 65–72. [CrossRef]

13. Alzaylaee, M.K.; Yerima, S.Y.; Sezer, S. DL-Droid: Deep learning based android malware detection using real devices. *Comput. Secur.* **2020**, *89*, 101663. [CrossRef]
14. Poeplau, S.; Fratantonio, Y.; Bianchi, A.; Kruegel, C.; Vigna, G. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In Proceedings of the Network and Distributed System Security Symposium, San Diego, CA, USA, 23–26 February 2014; Volume 14, pp. 23–26. [CrossRef]
15. Chen, K.; Wang, P.; Lee, Y.; Wang, X.; Zhang, N.; Huang, H.; Zou, W.; Liu, P. Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-Play Scale. In Proceedings of the SEC'15, 24th USENIX Conference on Security Symposium, Washington, DC, USA, 12–14 August 2015; pp. 659–674.
16. Arp, D.; Spreitzenbarth, M.; Hubner, M.; Gascon, H.; Rieck, K.; Siemens, C. Drebin: Effective and explainable detection of android malware in your pocket. In Proceedings of the Network and Distributed System Security Symposium, San Diego, CA, USA, 23–26 February 2014; Volume 14, pp. 23–26.
17. You, I.; Yim, K. Malware Obfuscation Techniques: A Brief Survey. In Proceedings of the 2010 International Conference on Broadband, Wireless Computing, Communication and Applications, Fukuoka, Japan, 4–6 November 2010; pp. 297–300. [CrossRef]
18. Sufatrio; Chua, T.W.; Tan, D.; Thing, V. Accurate Specification for Robust Detection of Malicious Behavior in Mobile Environments. In Proceedings of the European Symposium on Research in Computer Security, ESORICS 2015, Vienna, Austria, 21–25 September 2015; pp. 355–375. [CrossRef]
19. Sufatrio; Tan, D.J.J.; Chua, T.W.; Thing, V.L.L. Securing Android: A Survey, Taxonomy, and Challenges. *ACM Comput. Surv.* **2015**, *47*, 1–45. [CrossRef]
20. Xu, Z.; Zhang, J.; Gu, G.; Lin, Z. Goldeneye: Efficiently and effectively unveiling malware's targeted environment. In Proceedings of the International Workshop on Recent Advances in Intrusion Detection, Gothenburg, Sweden, 17–19 September 2014; pp. 22–45.
21. Rastogi, V.; Chen, Y.; Jiang, X. DroidChameleon: Evaluating Android Anti-Malware against Transformation Attacks. In Proceedings of the ASIA CCS '13, 8th ACM SIGSAC Symposium on Information, Computer and Communications Security, Hangzhou, China, 8–10 May 2013; pp. 329–334. [CrossRef]
22. Galloro, N.; Polino, M.; Carminati, M.; Continella, A.; Zanero, S. A Systematical and longitudinal study of evasive behaviors in windows malware. *Comput. Secur.* **2022**, *113*, 102550. [CrossRef]
23. Sihag, V.; Vardhan, M.; Singh, P. A survey of android application and malware hardening. *Comput. Sci. Rev.* **2021**, *39*, 100365. [CrossRef]
24. Jusoh, R.; Firdaus, A.; Anwar, S.; Osman, M.Z.; Darmawan, M.F.; Ab Razak, M.F. Malware detection using static analysis in Android: A review of FeCO (features, classification, and obfuscation). *PeerJ Comput. Sci.* **2021**, *7*, e522. [CrossRef] [PubMed]
25. Aslan, Ö.A.; Samet, R. A comprehensive review on malware detection approaches. *IEEE Access* **2020**, *8*, 6249–6271. [CrossRef]
26. Razgallah, A.; Khoury, R.; Hallé, S.; Khanmohammadi, K. A survey of malware detection in Android apps: Recommendations and perspectives for future research. *Comput. Sci. Rev.* **2021**, *39*, 100358. [CrossRef]
27. Chen, X.; Li, C.; Wang, D.; Wen, S.; Zhang, J.; Nepal, S.; Xiang, Y.; Ren, K. Android HIV: A study of repackaging malware for evading machine-learning detection. *IEEE Trans. Inf. Forensics Secur.* **2019**, *15*, 987–1001. [CrossRef]
28. Bhat, P.; Dutta, K. A Survey on Various Threats and Current State of Security in Android Platform. *ACM Comput. Surv.* **2019**, *52*, 21. [CrossRef]
29. Sen, S.; Aydogan, E.; Aysan, A.I. Coevolution of mobile malware and anti-malware. *IEEE Trans. Inf. Forensics Secur.* **2018**, *13*, 2563–2574. [CrossRef]
30. Dai, P.; Pan, Z.; Li, Y. A Review of Researching on Dynamic Taint Analysis Technique. In Proceedings of the 2018 3rd Joint International Information Technology, Mechanical and Electronic Engineering Conference (JIMEC 2018), Chongqing, China, 15–16 December 2018; pp. 118–123.
31. Xue, Y.; Meng, G.; Liu, Y.; Tan, T.H.; Chen, H.; Sun, J.; Zhang, J. Auditing anti-malware tools by evolving android malware and dynamic loading technique. *IEEE Trans. Inf. Forensics Secur.* **2017**, *12*, 1529–1544. [CrossRef]
32. Tam, K.; Feizollah, A.; Anuar, N.B.; Salleh, R.; Cavallaro, L. The Evolution of Android Malware and Android Analysis Techniques. *ACM Comput. Surv.* **2017**, *49*, 76. [CrossRef]
33. Nguyen-Vu, L.; Chau, N.T.; Kang, S.; Jung, S. Android rooting: An arms race between evasion and detection. *Secur. Commun. Netw.* **2017**, *2017*, 4121765. [CrossRef]
34. Preda, M.D.; Maggi, F. Testing android malware detectors against code obfuscation: A systematization of knowledge and unified methodology. *J. Comput. Virol. Hacking Tech.* **2017**, *13*, 209–232. [CrossRef]
35. Hoffmann, J.; Rytilahti, T.; Maiorca, D.; Winandy, M.; Giacinto, G.; Holz, T. Evaluating Analysis Tools for Android Apps: Status Quo and Robustness Against Obfuscation. In Proceedings of the CODASPY '16, Sixth ACM Conference on Data and Application Security and Privacy, Virtual, 26–28 April 2016; pp. 139–141. [CrossRef]
36. Kim, M.; Lee, T.J.; Shin, Y.; Youm, H.Y. A study on behavior-based mobile malware analysis system against evasion techniques. In Proceedings of the 2016 international conference on information networking (ICOIN), Kota Kinabalu, Malaysia, 13–15 January 2016; pp. 455–457.
37. Faruki, P.; Bharmal, A.; Laxmi, V.; Ganmoor, V.; Gaur, M.S.; Conti, M.; Rajarajan, M. Android security: A survey of issues, malware penetration, and defenses. *IEEE Commun. Surv. Tutor.* **2014**, *17*, 998–1022. [CrossRef]

38. Rastogi, V.; Chen, Y.; Jiang, X. Catch me if you can: Evaluating android anti-malware against transformation attacks. *IEEE Trans. Inf. Forensics Secur.* **2013**, *9*, 99–108. [CrossRef]
39. Diao, W.; Liu, X.; Li, Z.; Zhang, K. Evading android runtime analysis through detecting programmed interactions. In Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks, Darmstadt Germany, 18–20 July 2016; pp. 159–164.
40. Alaeiyan, M.; Parsa, S.; Conti, M. Analysis and classification of context-based malware behavior. *Comput. Commun.* **2019**, *136*, 76–90. [CrossRef]
41. Jang, D.; Jeong, Y.; Lee, S.; Park, M.; Kwak, K.; Kim, D.; Kang, B.B. Rethinking anti-emulation techniques for large-scale software deployment. *Comput. Secur.* **2019**, *83*, 182–200. [CrossRef]
42. Zhang, F.; Leach, K.; Stavrou, A.; Wang, H.; Sun, K. Using hardware features for increased debugging transparency. In Proceedings of the 2015 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 18–20 May 2015; pp. 55–69.
43. Choi, Y.; Jeong, Y.; Jang, D.; Kang, B.B.; Lee, H. EmuID: Detecting presence of emulation through microarchitectural characteristic on ARM. *Comput. Secur.* **2022**, *113*, 102569. [CrossRef]
44. Chen, K.H.; Shen, B.Y.; Yang, W. An automatic superword vectorization in LLVM. In Proceedings of the 16th Workshop on Compiler Techniques for High-Performance and Embedded Computing, Taipei, Taiwan, 27–28 May 2010; pp. 19–27.
45. Mayrhofer, R.; Stoep, J.V.; Brubaker, C.; Kralevich, N. The android platform security model. *ACM Trans. Priv. Secur. (TOPS)* **2021**, *24*, 1–35. [CrossRef]
46. Kirat, D.; Vigna, G.; Kruegel, C. {BareCloud}: Bare-metal Analysis-based Evasive Malware Detection. In Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14), San Diego, CA, USA, 20–22 August 2014; pp. 287–301.
47. Gründling, B. App-Based (Im) plausible Deniability for Android. Ph.D. Thesis, Johannes Kepler University Linz, Linz, Austria, 2020.
48. Arora, A.; Peddoju, S.K.; Conti, M. Permpair: Android malware detection using permission pairs. *IEEE Trans. Inf. Forensics Secur.* **2019**, *15*, 1968–1982. [CrossRef]
49. Lyvas, C. Security and Privacy Enhancing Mechanisms for the Android Operating System. Ph.D. Thesis, University of Piraeus, Pireas, Greece, 2021.
50. Lee, Y.T.; Chen, H.; Jaeger, T. Demystifying Android's Scoped Storage Defense. *IEEE Secur. Priv.* **2021**, *19*, 16–25. [CrossRef]
51. Heid, K.; Tefke, T.; Heider, J.; Staudemeyer, R.C. Android Data Storage Locations and What App Developers Do with It from a Security and Privacy Perspective. In Proceedings of the 8th International Conference on Information Systems Security and Privacy (ICISSP 2022), Online, 9–11 February 2022; pp. 378–387.
52. Abuthawabeh, M.K.A.; Mahmoud, K.W. Android malware detection and categorization based on conversation-level network traffic features. In Proceedings of the 2019 International Arab Conference on Information Technology (ACIT), Al Ain, United Arab Emirates, 3–5 December 2019; pp. 42–47.
53. Talos, C. PyREBox: Python scriptable reverse engineering sandbox. *Retrieved Aug* **2017**, *12*, 2018.
54. Zhou, Y.; Kim, D.W.; Zhang, J.; Liu, L.; Jin, H.; Jin, H.; Liu, T. Proguard: Detecting malicious accounts in social-network-based online promotions. *IEEE Access* **2017**, *5*, 1990–1999. [CrossRef]
55. Piao, Y.; Jung, J.H.; Yi, J.H. Server-based code obfuscation scheme for APK tamper detection. *Secur. Commun. Netw.* **2016**, *9*, 457–467. [CrossRef]
56. Palmaro, F. Evaluating Dynamic Binary Instrumentation Systems for Conspicuous Features and Artifacts. In *Digital Threats: Research and Practice*; Association for Computing Machinery: New York, NY, USA, 2022.
57. Li, X.; Li, K. Defeating the Transparency Features of Dynamic Binary Instrumentation. BlackHat USA. 2014. Available online: https://www.blackhat.com/docs/us-14/materials/us-14-Li-Defeating-The-Transparency-Feature-Of-DBI.pdf (accessed on 1 December 2022).
58. VirusTotal: An Alphabet Product That Analyzes Suspicious Files, URLs, Domains and IP Addresses. Available online: https://www.virustotal.com/gui/home (accessed on 1 December 2022).
59. Stolfo, S.; Wang, K.; Li, W.J. Towards Stealthy Malware Detection. In *Malware Detection*; Advances in Information Security; Springer: Boston, MA, USA, 2007; Volume 27, pp. 231–249.
60. Bunino, M. Reinforcement Learning-aided Dynamic Analysis of Evasive Malware. Master's Thesis, Politecnico di Torino, Torino, Italy, 2022.
61. Aafer, Y.; Du, W.; Yin, H. DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. In Proceedings of the SecureComm 2013, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, Sydney, Australia, 25–27 September 2013; Volume 127, pp. 86–103. [CrossRef]
62. Apostolopoulos, T.; Katos, V.; Choo, K.K.R.; Patsakis, C. Resurrecting anti-virtualization and anti-debugging: Unhooking your hooks. *Future Gener. Comput. Syst.* **2021**, *116*, 393–405. [CrossRef]
63. Kwon, J.; Jeong, J.; Lee, J.; Lee, H. DroidGraph: Discovering Android Malware by Analyzing Semantic Behavior. In Proceedings of the IEEE Conference on Communications and Network Security (CNS) 2014, San Francisco, CA, USA, 29–31 October 2014.
64. Crussell, J.; Gibler, C.; Chen, H. AnDarwin: Scalable Detection of Android Application Clones Based on Semantics. *IEEE Trans. Mob. Comput.* **2015**, *14*, 2007–2019. [CrossRef]

65. Li, L.; Bissyandé, T.F.; Octeau, D.; Klein, J. DroidRA: Taming Reflection to Support Whole-Program Analysis of Android Apps. In Proceedings of the ISSTA 2016, 25th International Symposium on Software Testing and Analysis, Saarbrucken, Germany, 18–20 July 2016; pp. 318–329. [CrossRef]
66. Zhang, F.; Leach, K.; Sun, K.; Stavrou, A. Spectre: A dependable introspection framework via system management mode. In Proceedings of the 2013 43rd Annual IEEE/IFIP international conference on dependable systems and networks (DSN), Budapest, Hungary, 24–27 June 2013; pp. 1–12.
67. Zhang, X.; Zhang, Y. ReACt: A Resource-centric Access Control System for Web-app Interactions on Android. In Proceedings of the Web Conference 2021, Online, 12–23 April 2021; pp. 1459–1470.
68. Chen, J.; Wang, C.; Zhao, Z.; Chen, K.; Du, R.; Ahn, G.J. Uncovering the face of android ransomware: Characterization and real-time detection. *IEEE Trans. Inf. Forensics Secur.* **2017**, *13*, 1286–1300. [CrossRef]
69. Zhauniarovich, Y.; Ahmad, M.; Gadyatskaya, O.; Crispo, B.; Massacci, F. StaDynA: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications. In Proceedings of the CODASPY '15, 5th ACM Conference on Data and Application Security and Privacy, San Antonio, TX, USA, 2–4 March 2015; pp. 37–48. [CrossRef]
70. Zhang, Y.; Luo, X.; Yin, H. DexHunter: Toward Extracting Hidden Code from Packed Android Applications. In Proceedings of the 20th European Symposium on Research in Computer Security, Vienna, Austria, 21–25 September 2015; pp. 293–311. [CrossRef]
71. Spreitzenbarth, M.; Freiling, F.; Echtler, F.; Schreck, T.; Hoffmann, J. Mobile-Sandbox: Having a Deeper Look into Android Applications. In Proceedings of the SAC '13, 28th Annual ACM Symposium on Applied Computing, Coimbra, Portugal, 18–22 March 2013; pp. 1808–1815. [CrossRef]
72. Jiang, M.; Xu, T.; Zhou, Y.; Hu, Y.; Zhong, M.; Wu, L.; Luo, X.; Ren, K. EXAMINER: Automatically locating inconsistent instructions between real devices and CPU emulators for ARM. In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 22 February 2022; pp. 846–858.
73. Oberheide, J.; Miller, C. Dissecting the android bouncer. *SummerCon2012* **2012**, *95*, 110.
74. Ma, H.; Li, S.; Gao, D.; Wu, D.; Jia, Q.; Jia, C. Active warden attack: On the (in) effectiveness of Android app repackage-proofing. *IEEE Trans. Dependable Secur. Comput.* **2021**, *19*, 3508–3520. [CrossRef]
75. VieiraB, B.; Rothermel, G.R.; Silva, E.; Bagheri, H.J. SEMEO: A Semantic Equivalence Analysis Framework for Obfuscated Android Applications. In Proceedings of the Mobile and Ubiquitous Systems: Computing, Networking and Services: 18th EAI International Conference, MobiQuitous 2021, Virtual Event, 8–11 November 2021; Volume 419, p. 322.
76. Wu, Y.; Dou, S.; Zou, D.; Yang, W.; Qiang, W.; Jin, H. Obfuscation-resilient Android malware analysis based on contrastive learning. *arXiv* **2021**, arXiv:2107.03799.
77. Cho, T.; Seo, S.H. A strengthened android signature management method. *KSII Trans. Internet Inf. Syst. (TIIS)* **2015**, *9*, 1210–1230.
78. Kim, Y.; Liszka, K.J.; Chan, C.C. Using DroidDream Android Malware Behavior for Identification of Other Android Malware Families. In Proceedings of the International Conference on Security and Management (SAM), Las Vegas, NV, USA, 25–28 July 2016; p. 286.
79. Baldoni, R.; Coppa, E.; D'Elia, D.C.; Demetrescu, C. Assisting malware analysis with symbolic execution: A case study. In Proceedings of the International Conference on Cyber Security Cryptography and Machine Learning, Beer-Sheva, Israel, 29–30 June 2017; pp. 171–188.
80. Schwartz, E.J.; Avgerinos, T.; Brumley, D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In Proceedings of the 2010 IEEE Symposium on Security and Privacy, Berleley/Oakland, CA, USA, 16–19 May 2010; pp. 317–331.
81. Baldoni, R.; Coppa, E.; D'elia, D.C.; Demetrescu, C.; Finocchi, I. A survey of symbolic execution techniques. *ACM Comput. Surv. (CSUR)* **2018**, *51*, 1–39. [CrossRef]
82. Borzacchiello, L.; Coppa, E.; D'Elia, D.C.; Demetrescu, C. Reconstructing C2 servers for remote access trojans with symbolic execution. In Proceedings of the International Symposium on Cyber Security Cryptography and Machine Learning, Beer-Sheva, Israel, 27–28 June 2019; pp. 121–140.
83. D'Onghia, M.; Salvadore, M.; Nespoli, B.M.; Carminati, M.; Polino, M.; Zanero, S. Apícula: Static Detection of API Calls in Generic Streams of Bytes. *Comput. Secur.* **2022**, *2022*, 102775. [CrossRef]
84. Yakdan, K.; Dechand, S.; Gerhards-Padilla, E.; Smith, M. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 21–24 March 2016; pp. 158–177.
85. Wei, F.; Li, Y.; Roy, S.; Ou, X.; Zhou, W. Deep Ground Truth Analysis of Current Android Malware. In *Detection of Intrusions and Malware, and Vulnerability Assessment*; Polychronakis, M., Meier, M., Eds.; Springer: Cham, Switzerland, 2017; pp. 252–276.
86. Palfinger, G.; Prünster, B.; Ziegler, D.J. AndroTIME: Identifying Timing Side Channels in the Android API. In Proceedings of the 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), Guangzhou, China, 10–13 November 2020; pp. 1849–1856.
87. Santos Filho, A.; Rodríguez, R.J.; Feitosa, E.L. Evasion and Countermeasures Techniques to Detect Dynamic Binary Instrumentation Frameworks. In *Digital Threats: Research and Practice*; Association for Computing Machinery: New York, NY, USA, 2022.

88. Bhan, R.; Pamula, R.; Faruki, P.; Gajrani, J. Blockchain-enabled secure and efficient data sharing scheme for trust management in healthcare smartphone network. *J. Supercomput.* 2023, *in press*. [CrossRef]
89. Zhou, Y.; Jiang, X. Dissecting Android Malware: Characterization and Evolution. In Proceedings of the 2012 IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 20–23 May 2012; pp. 95–109. [CrossRef]
90. Allix, K.; Bissyandé, T.F.; Klein, J.; Le Traon, Y. Androzoo: Collecting millions of android apps for the research community. In Proceedings of the 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), Austin, TX, USA, 14–15 May 2016; pp. 468–471.
91. Alahy, Q.E.; Chowdhury, M.; Soliman, H.; Chaity, M.S.; Haque, A. Android malware detection in large dataset: Smart approach. In Proceedings of the Future of Information and Communication Conference, San Francisco, CA, USA, 5–6 March 2020; pp. 800–814.
92. Glanz, L.; Amann, S.; Eichberg, M.; Reif, M.; Hermann, B.; Lerch, J.; Mezini, M. CodeMatch: Obfuscation Won'T Conceal Your Repackaged App. In Proceedings of the ESEC/FSE 2017, 2017 11th Joint Meeting on Foundations of Software Engineering, Paderborn, Germany, 4–8 September 2017; pp. 638–648. [CrossRef]
93. Canfora, G.; Medvet, E.; Mercaldo, F.; Visaggio, C.A. Acquiring and Analyzing App Metrics for Effective Mobile Malware Detection. In Proceedings of the IWSPA '16, 2016 ACM on International Workshop on Security And Privacy Analytics, New Orleans, LA, USA, 11 March 2016; pp. 50–57. [CrossRef]
94. Dolan-Gavitt, B.; Leek, T.; Zhivich, M.; Giffin, J.; Lee, W. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In Proceedings of the 2011 IEEE Symposium on Security and Privacy, Oakland, CA, USA, 22–25 May 2011; pp. 297–312.
95. Amin, M.; Tanveer, T.A.; Tehseen, M.; Khan, M.; Khan, F.A.; Anwar, S. Static malware detection and attribution in android byte-code through an end-to-end deep system. *Future Gener. Comput. Syst.* **2020**, *102*, 112–126. [CrossRef]
96. Cimato, S.; De Santis, A.; Ferraro Petrillo, U. Overcoming the obfuscation of Java programs by identifier renaming. *J. Syst. Software* **2005**, *78*, 60–72. [CrossRef]
97. Kirat, D.; Vigna, G. Malgene: Automatic extraction of malware analysis evasion signature. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, 12–16 October 2015; pp. 769–780.
98. Miramirkhani, N.; Appini, M.P.; Nikiforakis, N.; Polychronakis, M. Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts. In Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 22–26 May 2017; pp. 1009–1024.
99. Oltrogge, M. TLS on Android–Evolution over the Last Decade. Saarländische Universitäts-und Landesbibliothek. 2021. Available online: https://publikationen.sulb.uni-saarland.de/bitstream/20.500.11880/32875/1/thesis_final_Oltrogge.pdf (accessed on 1 December 2022).
100. Afifi, F.; Anuar, N.B.; Shamshirband, S.; Choo, K.K.R. DyHAP: Dynamic Hybrid ANFIS-PSO Approach for Predicting Mobile Malware. *PLoS ONE* **2016**, *11*, e0162627. [CrossRef]
101. Rafiq, H.; Aleem, M.; Islam, M.A. On the Evaluation of Android Malware Detectors: Evaluating Malware Detectors. *Sukkur IBA J. Comput. Math. Sci.* **2018**, *2*, 20–28.
102. Kalysch, A. *Android Application Hardening: Attack Surface Reduction and IP Protection Mechanisms*; Friedrich-Alexander-Universitaet Erlangen-Nuernberg: Erlangen, Germany, 2020.
103. Aonzo, S.; Georgiu, G.C.; Verderame, L.; Merlo, A. Obfuscapk: An open-source black-box obfuscation tool for Android apps. *SoftwareX* **2020**, *11*, 100403. [CrossRef]
104. Franke, B. Fast Cycle-Approximate Instruction Set Simulation. In Proceedings of the SCOPES '08, 11th International Workshop on Software & Compilers for Embedded Systems, Munich Germany, 13–14 March 2008; pp. 69–78. [CrossRef]
105. Alrabaee, S.; Debbabi, M.; Wang, L. A Survey of Binary Code Fingerprinting Approaches: Taxonomy, Methodologies, and Features. *ACM Comput. Surv. (CSUR)* **2022**, *55*, 1–41. [CrossRef]
106. Elsersy, W.F.; Feizollah, A.; Anuar, N.B. The rise of obfuscated Android malware and impacts on detection methods. *PeerJ Comput. Sci.* **2022**, *8*, e907. [CrossRef]
107. Muralidharan, T.; Cohen, A.; Gerson, N.; Nissim, N. File Packing from the Malware Perspective: Techniques, Analysis Approaches, and Directions for Enhancements. *ACM Comput. Surv.* **2022**, *55*, 108. [CrossRef]
108. Elgharabawy, M. Cross-vendor Security Analysis of Android Unix Domain Sockets. Ph.D. Thesis, Concordia University, Montréal, QC, Canada, 2021.
109. Filho, A.S.; Rodríguez, R.J.; Feitosa, E.L. Evasion and Countermeasures Techniques to Detect Dynamic Binary Instrumentation Frameworks. *Digit. Threat. Res. Pract.e (DTRAP)* **2022**, *3*, 1–28. [CrossRef]
110. Lau, B.; Svajcer, V. Measuring virtual machine detection in malware using DSD tracer. *J. Comput. Virol.* **2010**, *6*, 181–195. [CrossRef]
111. Omella, A. Methods for Virtual Machine Detection. Grupo S21sec Gestión SA. 2006. Available online: https://www.s21sec.com/ (accessed on 1 December 2022).
112. Wang, J.B.; Lian, Y.F.; Chen, K. Virtualization detection based on data fusion. In Proceedings of the 2012 International Conference on Computer Science and Information Processing (CSIP), Xi'an, China, 24–26 August 2012; pp. 393–396. [CrossRef]
113. Agman, Y.; Hendler, D. BPFroid: Robust Real Time Android Malware Detection Framework. *arXiv* **2021**, arXiv:2105.14344.
114. Yan, L.K.; Yin, H. {DroidScope}: Seamlessly Reconstructing the {OS} and Dalvik Semantic Views for Dynamic Android Malware Analysis. In Proceedings of the 21st USENIX Security Symposium (USENIX Security 12), Bellevue, WA, USA, 8–10 August 2012; pp. 569–584.

115. Petsas, T.; Voyatzis, G.; Athanasopoulos, E.; Polychronakis, M.; Ioannidis, S. Rage against the virtual machine: Hindering dynamic analysis of android malware. In Proceedings of the Seventh European Workshop on System Security, Amsterdam, The Netherlands, 13–16 April 2014; pp. 1–6.
116. Sharma, A.; Gupta, B.B.; Singh, A.K.; Saraswat, V. Orchestration of APT malware evasive manoeuvers employed for eluding anti-virus and sandbox defense. *Comput. Secur.* **2022**, *115*, 102627. [CrossRef]
117. Qu, Z.; Alam, S.; Chen, Y.; Zhou, X.; Hong, W.; Riley, R. DyDroid: Measuring Dynamic Code Loading and Its Security Implications in Android Applications. In Proceedings of the 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Denver, CO, USA, 26–29 June 2017; pp. 415–426. [CrossRef]
118. Zhang, F.; Leach, K.; Stavrou, A.; Wang, H. Towards transparent debugging. *IEEE Trans. Dependable Secur. Comput.* **2016**, *15*, 321–335. [CrossRef]
119. Shi, H.; Mirkovic, J. Hiding debuggers from malware with apate. In Proceedings of the Symposium on Applied Computing, Marrakech, Morocco, 4–6 April 2017; pp. 1703–1710.
120. Lindorfer, M.; Kolbitsch, C.; Milani Comparetti, P. Detecting environment-sensitive malware. In Proceedings of the International Workshop on Recent Advances in Intrusion Detection, Menlo Park, CA, USA, 20–21 September 2011; pp. 338–357.
121. Ferrie, P. The Ultimate Anti-Debugging Reference. Available online: https://www.anti-reversing.com/ (accessed on 1 December 2022).
122. Sinha, A.; Di Troia, F.; Heller, P.; Stamp, M. Emulation Versus Instrumentation for Android Malware Detection. In *Digital Forensic Investigation of Internet of Things (IoT) Devices*; Springer: Berlin/Heidelberg, Germany, 2021; pp. 1–20.
123. Conley, J.; Andros, E.; Chinai, P.; Lipkowitz, E. Use of a game over: Emulation and the video game industry, a white paper. *Nw. J. Tech. Intell. Prop.* **2003**, *2*, 261.
124. Lee, S. A study on android emulator detection for mobile game security. *J. Korea Inst. Inf. Secur. Cryptol.* **2014**, *25*, 1067–1075.
125. Jing, Y.; Zhao, Z.; Ahn, G.J.; Hu, H. Morpheus: Automatically Generating Heuristics to Detect Android Emulators. In Proceedings of the ACSAC '14, 30th Annual Computer Security Applications Conference, New Orleans, LA, USA, 8–12 December 2014; pp. 216–225. [CrossRef]
126. Shi, H.; Mirkovic, J.; Alwabel, A. Handling anti-virtual machine techniques in malicious software. *ACM Trans. Priv. Secur. (TOPS)* **2017**, *21*, 1–31. [CrossRef]
127. Brengel, M.; Backes, M.; Rossow, C. Detecting hardware-assisted virtualization. In Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, San Sebastian, Spain, 7–8 July 2016; pp. 207–227.
128. Oyama, Y. How does malware use RDTSC? A study on operations executed by malware with CPU cycle measurement. In Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Gothenburg, Sweden, 19–20 June 2019; pp. 197–218.
129. D'Elia, D.C.; Coppa, E.; Nicchi, S.; Palmaro, F.; Cavallaro, L. SoK: Using dynamic binary instrumentation for security (and how you may get caught red handed). In Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, Auckland, New Zealand, 9–12 July 2019; pp. 15–27.
130. Peng, F.; Deng, Z.; Zhang, X.; Xu, D.; Lin, Z.; Su, Z. {X-Force}:{Force-Executing} Binary Programs for Security Applications. In Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14), San Diego, CA, USA, 20–22 August 2014; pp. 829–844.
131. Sun, K.; Li, X.; Ou, Y. Break Out of the Truman Show: Active Detection and Escape of Dynamic Binary Instrumentation. Black Hat Asia. 2016. Available online: https://www.blackhat.com/us-16/briefings/schedule/ (accessed on 1 December 2022).
132. Kirsch, J.; Zhechev, Z.; Bierbaumer, B.; Kittel, T. PwIN–Pwning Intel piN: Why DBI is unsuitable for security applications. In Proceedings of the European Symposium on Research in Computer Security, Copenhagen, Denmark, 26–30 September 2018; pp. 363–382.
133. Lee, Y.B.; Suk, J.H.; Lee, D.H. Bypassing Anti-Analysis of Commercial Protector Methods Using DBI Tools. *IEEE Access* **2021**, *9*, 7655–7673. [CrossRef]
134. Nethercote, N.; Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Not.* **2007**, *42*, 89–100. [CrossRef]
135. Paleari, R.; Martignoni, L.; Roglia, G.F.; Bruschi, D. A fistful of red-pills: How to automatically generate procedures to detect CPU emulators. In Proceedings of the USENIX Workshop on Offensive Technologies (WOOT), Montreal, QC Canada, 10 August 2009; Volume 41, p. 86.
136. Jiang, M.; Xu, T.; Zhou, Y.; Hu, Y.; Zhong, M.; Wu, L.; Luo, X.; Ren, K. Automatically Locating ARM Instructions Deviation between Real Devices and CPU Emulators. *arXiv* **2021**, arXiv:2105.14273.
137. Dinaburg, A.; Royal, P.; Sharif, M.; Lee, W. Ether: Malware Analysis via Hardware Virtualization Extensions. In Proceedings of the CCS '08, 15th ACM conference on Computer and Communications Security, Alexandria, VA, USA, 27–31 October 2008; pp. 51–62. [CrossRef]
138. Liu, W.; Liu, X.; Li, Z.; Liu, B.; Yu, R.; Wang, L. Retrofitting LBR Profiling to Enhance Virtual Machine Introspection. *IEEE Trans. Inf. Forensics Secur.* **2022**, *17*, 2311–2323. [CrossRef]
139. Melvin, A.A.R.; Kathrine, G.J.W.; Ilango, S.S.; Vimal, S.; Rho, S.; Xiong, N.N.; Nam, Y. Dynamic malware attack dataset leveraging virtual machine monitor audit data for the detection of intrusions in cloud. *Trans. Emerg. Telecommun. Technol.* **2022**, *33*, e4287. [CrossRef]

140. D'Elia, D.C.; Coppa, E.; Palmaro, F.; Cavallaro, L. On the dissection of evasive malware. *IEEE Trans. Inf. Forensics Secur.* **2020**, *15*, 2750–2765. [CrossRef]
141. Wu, D.J.; Mao, C.H.; Wei, T.E.; Lee, H.M.; Wu, K.P. DroidMat: Android Malware Detection through Manifest and API Calls Tracing. In Proceedings of the 2012 Seventh Asia Joint Conference on Information Security, Tokyo, Japan, 9–10 August 2012; pp. 62–69. [CrossRef]
142. Zhou, W.; Zhou, Y.; Jiang, X.; Ning, P. Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces. In Proceedings of the CODASPY '12, Second ACM Conference on Data and Application Security and Privacy, San Antonio, TX, USA, 7–9 February 2012; pp. 317–326. [CrossRef]
143. Sanz, B.; Santos, I.; Laorden, C.; Ugarte-Pedrero, X.; Nieves, J.; Bringas, P.G.; Álvarez Marañón, G. MAMA: Manifest Analysis for Malware Detection in Android. *Cybern. Syst.* **2013**, *44*, 469–488. [CrossRef]
144. Markmann, T.; Gessner, D.; Westhoff, D. QuantDroid: Quantitative approach towards mitigating privilege escalation on Android. In Proceedings of the 2013 IEEE International Conference on Communications (ICC), Budapest, Hungary, 9–13 June 2013; pp. 2144–2149. [CrossRef]
145. Suarez-Tangil, G.; Tapiador, J.E.; Peris-Lopez, P.; Blasco, J. Dendroid: A text mining approach to analyzing and classifying code structures in Android malware families. *Expert Syst. Appl.* **2014**, *41*, 1104–1117. [CrossRef]
146. Sheen, S.; Ramalingam, A. Malware detection in Android files based on multiple levels of learning and diverse data sources. In Proceedings of the Third International Symposium on Women in Computing and Informatics, Kerala, India, 10–13 August 2015; pp. 553–559.
147. Talha, K.A.; Alper, D.I.; Aydin, C. APK Auditor: Permission-based Android malware detection system. *Digit. Investig.* **2015**, *13*, 1–14. [CrossRef]
148. Zhang, X.; Breitinger, F.; Baggili, I. Rapid Android Parser for Investigating DEX files (RAPID). *Digit. Investig.* **2016**, *17*, 28–39. [CrossRef]
149. Grace, M.; Zhou, Y.; Zhang, Q.; Zou, S.; Jiang, X. RiskRanker: Scalable and Accurate Zero-Day Android Malware Detection. In Proceedings of the MobiSys '12, 10th International Conference on Mobile Systems, Applications, and Services, Ambleside, UK, 25–29 June 2012; pp. 281–294. [CrossRef]
150. Xu, J.; Yu, Y.; Chen, Z.; Cao, B.; Dong, W.; Guo, Y.; Cao, J. MobSafe: Cloud computing based forensic analysis for massive mobile applications using data mining. *Tsinghua Sci. Technol.* **2013**, *18*, 418–427. [CrossRef]
151. Faruki, P.; Laxmi, V.; Ganmoor, V.; Gaur, M.S.; Bharmal, A. DroidOLytics: Robust Feature Signature for Repackaged Android Apps on Official and Third Party Android Markets. In Proceedings of the 2013 2nd International Conference on Advanced Computing, Networking and Security, Mangalore, India, 15–17 December 2013; pp. 247–252. [CrossRef]
152. Amos, B.; Turner, H.; White, J. Applying machine learning classifiers to dynamic Android malware detection at scale. In Proceedings of the 2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC), Sardinia, Italy, 1–5 July 2013; pp. 1666–1671.
153. Maggi, F.; Valdi, A.; Zanero, S. AndroTotal: A Flexible, Scalable Toolbox and Service for Testing Mobile Malware Detectors. In Proceedings of the SPSM '13, Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices, Berlin, Germany, 8 November 2013; pp. 49–54. [CrossRef]
154. Shalaginov, A.; Franke, K. Automatic rule-mining for malware detection employing neuro-fuzzy approach. In Proceedings of the Norsk Informasjonssikkerhetskonferanse (NISK), Stavanger, Norway, 18–20 November 2013.
155. Zhong, Y.; Yamaki, H.; Yamaguchi, Y.; Takakura, H. ARIGUMA Code Analyzer: Efficient Variant Detection by Identifying Common Instruction Sequences in Malware Families. In Proceedings of the 2013 IEEE 37th Annual Computer Software and Applications Conference (COMPSAC), Kyoto, Japan, 22–26 July 2013; pp. 11–20. [CrossRef]
156. Zhang, F.; Huang, H.; Zhu, S.; Wu, D.; Liu, P. ViewDroid: Towards Obfuscation-Resilient Mobile Application Repackaging Detection. In Proceedings of the WiSec '14, 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks, Oxford, UK, 23–25 July 2014; pp. 25–36. [CrossRef]
157. Jeong, J.; Seo, D.; Lee, C.; Kwon, J.; Lee, H.; Milburn, J. MysteryChecker: Unpredictable attestation to detect repackaged malicious applications in Android. In Proceedings of the 2014 9th International Conference on Malicious and Unwanted Software: The Americas (MALWARE), Fajardo, PR, USA, 28–30 October 2014; pp. 50–57. [CrossRef]
158. Shao, Y.; Luo, X.; Qian, C.; Zhu, P.; Zhang, L. Towards a Scalable Resource-Driven Approach for Detecting Repackaged Android Applications. In Proceedings of the ACSAC '14, 30th Annual Computer Security Applications Conference, New Orleans, LA, USA, 8–12 December 2014; pp. 56–65. [CrossRef]
159. Enck, W.; Gilbert, P.; Han, S.; Tendulkar, V.; Chun, B.G.; Cox, L.P.; Jung, J.; McDaniel, P.; Sheth, A.N. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Trans. Comput. Syst.* **2014**, *32*, 1–29. [CrossRef]
160. Pektaş, A.; Acarman, T. A dynamic malware analyzer against virtual machine aware malicious software. *Secur. Commun. Netw.* **2014**, *7*, 2245–2257. [CrossRef]
161. Soh, C.; Tan, H.B.K.; Arnatovich, Y.L.; Wang, L. Detecting Clones in Android Applications through Analyzing User Interfaces. In Proceedings of the ICPC '15, 2015 IEEE 23rd International Conference on Program Comprehension, Florence, Italy, 18–19 May 2015; pp. 163–173.
162. Shabtai, A.; Tenenboim-Chekina, L.; Mimran, D.; Rokach, L.; Shapira, B.; Elovici, Y. Mobile malware detection through analysis of deviations in application network behavior. *Comput. Secur.* **2014**, *43*, 1–18. [CrossRef]

163. Zhang, Y.; Yang, M.; Xu, B.; Yang, Z.; Gu, G.; Ning, P.; Wang, X.S.; Zang, B. Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis. In Proceedings of the CCS '13, 2013 ACM SIGSAC Conference on Computer & Communications Security, Hangzhou, China, 8–10 May 2013; pp. 611–622. [CrossRef]
164. Almohri, H.M.; Yao, D.D.; Kafura, D. DroidBarrier: Know What is Executing on Your Android. In Proceedings of the CODASPY '14, 4th ACM Conference on Data and Application Security and Privacy, San Antonio, TX, USA, 3–5 March 2014; pp. 257–264. [CrossRef]
165. Bengio, Y. Learning Deep Architectures for AI. *Found. Trends Mach. Learn.* **2009**, *2*, 1–127. [CrossRef]
166. Zhao, S.; Li, X.; Xu, G.; Zhang, L.; Feng, Z. Attack Tree Based Android Malware Detection with Hybrid Analysis. In Proceedings of the 2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications, Beijing, China, 24–26 September 2014; pp. 380–387.
167. Reddy, D.; Pujari, A.K. N-gram analysis for computer virus detection. *J. Comput. Virol.* **2006**, *2*, 231–239. [CrossRef]
168. Wei, T.E.; Tyan, H.R.; Jeng, A.B.; Lee, H.M.; Liao, H.Y.M.; Wang, J.C. DroidExec: Root exploit malware recognition against wide variability via folding redundant function-relation graph. In Proceedings of the 2015 17th International Conference on Advanced Communication Technology (ICACT), PyeongChang, Republic of Korea, 1–3 July 2015; pp. 161–169. [CrossRef]
169. Faruki, P.; Laxmi, V.; Bharmal, A.; Gaur, M.; Ganmoor, V. AndroSimilar: Robust signature for detecting variants of Android malware. *J. Inf. Secur. Appl.* **2015**, *22*, 66–80.
170. Canfora, G.; De Lorenzo, A.; Medvet, E.; Mercaldo, F.; Visaggio, C.A. Effectiveness of opcode ngrams for detection of multi family android malware. In Proceedings of the 2015 10th International Conference on Availability, Reliability and Security, Toulouse, France, 24–27 August 2015; pp. 333–340.
171. Sun, M.; Li, M.; Lui, J.C.S. DroidEagle: Seamless Detection of Visually Similar Android Apps. In Proceedings of the WiSec '15, 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks, New York, NY, USA, 22–26 June 2015. [CrossRef]
172. Gonzalez, H.; Stakhanova, N.; Ghorbani, A.A. DroidKin: Lightweight Detection of Android Apps Similarity. In Proceedings of the 10th International ICST Conference, SecureComm 2014, Beijing, China, 24–26 September 2014; volume 152, pp. 436–453. [CrossRef]
173. Shen, T.; Zhongyang, Y.; Xin, Z.; Mao, B.; Huang, H. Detect android malware variants using component based topology graph. In Proceedings of the 2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications, Beijing, China, 24–26 September 2014; pp. 406–413.
174. Apvrille, A.; Apvrille, L. SherlockDroid: A research assistant to spot unknown malware in Android marketplaces. *J. Comput. Virol. Hacking Tech.* **2015**, *11*, 235–245. [CrossRef]
175. Kühnel, M.; Smieschek, M.; Meyer, U. Fast identification of obfuscation and mobile advertising in mobile malware. In Proceedings of the 2015 IEEE Trustcom/BigDataSE/ISPA, Helsinki, Finland, 20–22 August 2015; Volume 1, pp. 214–221.
176. Salva, S.; Zafimiharisoa, S.R. APSET, an Android aPplication SEcurity Testing tool for detecting intent-based vulnerabilities. *Int. J. Softw. Tools Technol. Transf.* **2015**, *17*, 201–221. [CrossRef]
177. Afonso, V.M.; de Amorim, M.F.; Grégio, A.R.A.; Junquera, G.B.; de Geus, P.L. Identifying Android malware using dynamically obtained features. *J. Comput. Virol. Hacking Tech.* **2015**, *11*, 9–17. [CrossRef]
178. Maier, D.; Protsenko, M.; Müller, T. A game of Droid and Mouse: The threat of split-personality malware on Android. *Comput. Secur.* **2015**, *54*, 2–15.
179. Singh, S.; Mishra, B.; Singh, S. Detecting intelligent malware on dynamic Android analysis environments. In Proceedings of the 2015 10th International Conference for Internet Technology and Secured Transactions (ICITST), London, UK, 14–16 December 2015; pp. 414–419.
180. Gheorghe, L.; Marin, B.; Gibson, G.; Mogosanu, L.; Deaconescu, R.; Voiculescu, V.G.; Carabas, M. Smart malware detection on Android. *Secur. Commun. Netw.* **2015**, *8*, 4254–4272. [CrossRef]
181. Kim, D.; Kwak, J.; Ryou, J. DWroidDump: Executable Code Extraction from Android Applications for Malware Analysis. *Int. J. Distrib. Sens. Netw.* **2015**, *11*, 379682.
182. Wang, C.; Hwang, J.G. Automatic clustering using particle swarm optimization with various validity indices. In Proceedings of the 5th International Conference on BioMedical Engineering and Informatics, BMEI 2012, Chongqing, China, 16–18 October 2012; pp. 1557–1561. [CrossRef]
183. Abraham, A.; Andriatsimandefitra, R.; Brunelat, A.; Lalande, J.F.; Viet Triem Tong, V. GroddDroid: A Gorilla for Triggering Malicious Behaviors. In Proceedings of the 10th International Conference on Malicious and Unwanted Software, Fajardo, PR, USA, 20–22 October 2015.
184. Wu, X.; Zhang, D.; Su, X.; Li, W. Detect repackaged android application based on http traffic similarity. *Secur. Commun. Netw.* **2015**, *8*, 2257–2266. [CrossRef]
185. Lindorfer, M.; Neugschwandtner, M.; Platzer, C. MARVIN: Efficient and Comprehensive Mobile App Classification through Static and Dynamic Analysis. In Proceedings of the 2015 IEEE 39th Annual Computer Software and Applications Conference, Taichung, Taiwan, 1–5 July 2015; Volume 2, pp. 422–433. [CrossRef]
186. Shrestha, B.; Ma, D.; Zhu, Y.; Li, H.; Saxena, N. Tap-Wave-Rub: Lightweight Human Interaction Approach to Curb Emerging Smartphone Malware. *IEEE Trans. Inf. Forensics Secur.* **2015**, *10*, 2270–2283. [CrossRef]
187. Gurulian, I.; Markantonakis, K.; Cavallaro, L.; Mayes, K. You can't touch this: Consumer-centric android application repackaging detection. *Future Gener. Comput. Syst.* **2016**, *65*, 1–9. [CrossRef]

188. Fratantonio, Y.; Bianchi, A.; Robertson, W.; Kirda, E.; Kruegel, C.; Vigna, G. TriggerScope: Towards Detecting Logic Bombs in Android Applications. In Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 21–24 March 2016; pp. 377–396. [CrossRef]
189. Alejandro, M.; Menendez, H.D.; Camacho, D. MOCDroid: Multi-objective evolutionary classifier for Android malware detection. *Soft Comput.* **2017**, *21*, 7405–7415. [CrossRef]
190. Battista, P.; Mercaldo, F.; Nardone, V.; Santone, A.; Visaggio, C.A. Identification of Android Malware Families with Model Checking. In Proceedings of the 2nd International Conference on Information Systems Security and Privacy—Volume 1: ICISSP. INSTICC, SciTePress, Rome, Italy, 19–21 February 2016; pp. 542–547. [CrossRef]
191. Alzaylaee, M.K.; Yerima, S.Y.; Sezer, S. Dynalog: An automated dynamic analysis framework for characterizing android applications. In Proceedings of the 2016 International Conference on Cyber Security And Protection OF Digital Services (Cyber Security), London, UK, 13–14 June 2016; pp. 1–8. [CrossRef]
192. Castellanos, J.H.; Wuchner, T.; Ochoa, M.; Rueda, S. Q-Floid: Android Malware detection with Quantitative Data Flow Graphs. In Proceedings of the Singapore Cyber-Security Conference (SG-CRC), Singapore, 14–15 January 2016; Volume 14, pp. 13–25. [CrossRef]
193. Yuan, Z.; Lu, Y.; Xue, Y. Droiddetector: Android malware characterization and detection using deep learning. *Tsinghua Sci. Technol.* **2016**, *21*, 114–123. [CrossRef]
194. Jang, J.W.; Kang, H.; Woo, J.; Mohaisen, A.; Kim, H.K. Andro-Dumpsys: Anti-malware system based on the similarity of malware creator and malware centric information. *Comput. Secur.* **2016**, *58*, 125–138. [CrossRef]
195. Suarez-Tangil, G.; Dash, S.K.; Ahmadi, M.; Kinder, J.; Giacinto, G.; Cavallaro, L. Droidsieve: Fast and accurate classification of obfuscated android malware. In Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, Scottsdale, AZ, USA, 22–24 March 2017; pp. 309–320.
196. Titze, D.; Lux, M.; Schuette, J. Ordol: Obfuscation-resilient detection of libraries in android applications. In Proceedings of the 2017 IEEE Trustcom/BigDataSE/ICESS, Sydney, Australia, 1–4 August 2017; pp. 618–625.
197. Khanmohammadi, K.; Hamou-Lhadj, A. Hydroid: A hybrid approach for generating API Call traces from obfuscated android applications for mobile security. In Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), Hainan, China, 6–10 December 2017; pp. 168–175.
198. Bello, L.; Pistoia, M. Ares: Triggering payload of evasive android malware. In Proceedings of the 2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft), Gothenburg, Sweden, 27–28 May 2018; pp. 2–12.
199. Bacci, A.; Bartoli, A.; Martinelli, F.; Medvet, E.; Mercaldo, F.; Visaggio, C.A. Impact of Code Obfuscation on Android Malware Detection based on Static and Dynamic Analysis. In Proceedings of the 4th International Conference on Information Systems Security and Privacy, Madeira, Portugal, 22–24 January 2018; pp. 379–385.
200. Cai, H.; Meng, N.; Ryder, B.; Yao, D. Droidcat: Effective android malware detection and categorization via app-level profiling. *IEEE Trans. Inf. Forensics Secur.* **2018**, *14*, 1455–1470. [CrossRef]
201. Mirzaei, O.; de Fuentes, J.M.; Tapiador, J.; Gonzalez-Manzano, L. AndrODet: An adaptive Android obfuscation detector. *Future Gener. Comput. Syst.* **2019**, *90*, 240–261. [CrossRef]
202. Ikram, M.; Beaume, P.; Kâafar, M.A. Dadidroid: An obfuscation resilient tool for detecting android malware via weighted directed call graph modelling. *arXiv* **2019**, arXiv:1905.09136.
203. Li, Z.; Sun, J.; Yan, Q.; Srisa-an, W.; Tsutano, Y. Obfusifier: Obfuscation-resistant android malware detection system. In Proceedings of the International Conference on Security and Privacy in Communication Systems, Orlando, FL, USA, 23–25 October 2019; pp. 214–234.
204. Kim, T.; Kang, B.; Rho, M.; Sezer, S.; Im, E.G. A Multimodal Deep Learning Method for Android Malware Detection Using Various Features. *IEEE Trans. Inf. Forensics Secur.* **2019**, *14*, 773–788. [CrossRef]
205. Alazab, M.; Alazab, M.; Shalaginov, A.; Mesleh, A.; Awajan, A. Intelligent mobile malware detection using permission requests and API calls. *Future Gener. Comput. Syst.* **2020**, *107*, 509–521. [CrossRef]
206. Zhang, W.; Wang, H.; He, H.; Liu, P. DAMBA: Detecting android malware by ORGB analysis. *IEEE Trans. Reliab.* **2020**, *69*, 55–69. [CrossRef]
207. Vasan, D.; Alazab, M.; Wassan, S.; Naeem, H.; Safaei, B.; Zheng, Q. IMCFN: Image-based malware classification using fine-tuned convolutional neural network architecture. *Comput. Netw.* **2020**, *171*, 107138. [CrossRef]
208. Alrzini, J.R.S.; Pennington, D. A review of polymorphic malware detection techniques. *Int. J. Adv. Res. Eng. Technol.* **2020**, *11*, 1238–1247.
209. Karbab, E.B.; Debbabi, M. Resilient and adaptive framework for large scale android malware fingerprinting using deep learning and NLP techniques. *arXiv* **2021**, arXiv:2105.13491.
210. Sihag, V.; Vardhan, M.; Singh, P. BLADE: Robust malware detection against obfuscation in android. *Forensic Sci. Int. Digit. Investig.* **2021**, *38*, 301176. [CrossRef]
211. Dharmalingam, V.P.; Palanisamy, V. A novel permission ranking system for android malware detection—the permission grader. *J. Ambient. Intell. Humaniz. Comput.* **2021**, *12*, 5071–5081. [CrossRef]
212. Zou, D.; Wu, Y.; Yang, S.; Chauhan, A.; Yang, W.; Zhong, J.; Dou, S.; Jin, H. IntDroid: Android malware detection based on API intimacy analysis. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **2021**, *30*, 1–32. [CrossRef]

213. Molina-Coronado, B.; Mori, U.; Mendiburu, A.; Miguel-Alonso, J. Towards a fair comparison and realistic evaluation framework of android malware detectors based on static analysis and machine learning. *Comput. Secur.* **2023**, *124*, 102996. [CrossRef]
214. D'Elia, D.C.; Nicchi, S.; Mariani, M.; Marini, M.; Palmaro, F. Designing robust API monitoring solutions. *IEEE Trans. Dependable Secur. Comput.* **2021**, *20*, 392–406. [CrossRef]
215. Gajrani, J.; Laxmi, V.; Tripathi, M.; Gaur, M.S.; Zemmari, A.; Mosbah, M.; Conti, M. Effectiveness of state-of-the-art dynamic analysis techniques in identifying diverse Android malware and future enhancements. In *Advances in Computers*; Elsevier: Amsterdam, The Netherlands, 2020; Volume 119, pp. 73–120.
216. Qian, C.; Luo, X.; Le, Y.; Gu, G. VulHunter: Toward Discovering Vulnerabilities in Android Applications. *IEEE Micro* **2015**, *35*, 44–53. [CrossRef]
217. Kang, M.G.; Yin, H.; Hanna, S.; McCamant, S.; Song, D. Emulating emulation-resistant malware. In Proceedings of the 1st ACM Workshop on Virtual Machine Security, Chicago, IL, USA, 13 November 2009; pp. 11–22.
218. Kawakoya, Y.; Iwamura, M.; Shioji, E.; Hariu, T. Api chaser: Anti-analysis resistant malware analyzer. In Proceedings of the International Workshop on Recent Advances in Intrusion Detection, Rodney Bay, St. Lucia, 23–25 October 2013; pp. 123–143.
219. Ahn, N.Y.; Lee, D.H. Forensics and anti-forensics of a NAND flash memory: From a copy-back program perspective. *IEEE Access* **2021**, *9*, 14130–14137. [CrossRef]
220. Chaugule, A.; Xu, Z.; Zhu, S. A Specification Based Intrusion Detection Framework for Mobile Phones. In Proceedings of the ACNS'11, 9th International Conference on Applied Cryptography and Network Security, Nerja, Spain, 7–10 June 2011; pp. 19–37.
221. Feng, T.; Liu, Z.; Kwon, K.A.; Shi, W.; Carbunar, B.; Jiang, Y.; Nguyen, N. Continuous mobile authentication using touchscreen gestures. In Proceedings of the 2012 IEEE Conference on Technologies for Homeland Security (HST), Waltham, MA, USA, 13–15 November 2012; pp. 451–456.
222. Shabtai, A.; Kanonov, U.; Elovici, Y.; Glezer, C.; Weiss, Y. Andromaly: A behavioral malware detection framework for android devices. *J. Intell. Inf. Syst.* **2012**, *38*, 161–190. [CrossRef]
223. Glodek, W.; Harang, R. Rapid Permissions-Based Detection and Analysis of Mobile Malware Using Random Decision Forests. In Proceedings of the MILCOM 2013—2013 IEEE Military Communications Conference, San Diego, CA, USA, 18–20 November 2013; pp. 980–985. [CrossRef]
224. Yerima, S.Y.; Sezer, S.; McWilliams, G. Analysis of Bayesian classification-based approaches for Android malware detection. *IET Inf. Secur.* **2014**, *8*, 25–36. [CrossRef]
225. Narayanan, A.; Chen, L.; Chan, C.K. AdDetect: Automated detection of Android ad libraries using semantic analysis. In Proceedings of the 2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), Singapore, 21–24 April 2014; pp. 1–6. [CrossRef]
226. Feng, P.; Ma, J.; Sun, C.; Xu, X.; Ma, Y. A novel dynamic android malware detection system with ensemble learning. *IEEE Access* **2018**, *6*, 30996–31011. [CrossRef]
227. Wu, Y.; Zou, D.; Yang, W.; Li, X.; Jin, H. HomDroid: Detecting Android covert malware by social-network homophily analysis. In Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual, 11–17 July 2021; pp. 216–229.
228. Liu, P.; Wang, W.; Luo, X.; Wang, H.; Liu, C. NSDroid: Efficient multi-classification of android malware using neighborhood signature in local function call graphs. *Int. J. Inf. Secur.* **2021**, *20*, 59–71. [CrossRef]
229. Ou, F.; Xu, J. S3Feature: A static sensitive subgraph-based feature for android malware detection. *Comput. Secur.* **2022**, *112*, 102513. [CrossRef]
230. Elsersy, W.F.; Anuar, N.B.; Razak, M.F.A. ROOTECTOR: Robust Android Rooting Detection Framework Using Machine Learning Algorithms. *Arab. J. Sci. Eng.* **2023**, *48*, 1771–1791. [CrossRef]
231. Wei, F.; Roy, S.; Ou, X.; Robby. Amandroid: A Precise and General Inter-Component Data Flow Analysis Framework for Security Vetting of Android Apps. In Proceedings of the CCS '14, 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, 3–7 November 2014; pp. 1329–1341. [CrossRef]
232. Gao, T.; Peng, W.; Sisodia, D.; Saha, T.K.; Li, F.; Al Hasan, M. Android Malware Detection via Graphlet Sampling. *IEEE Trans. Mob. Comput.* **2019**, *18*, 2754–2767. [CrossRef]
233. Leach, K.; Spensky, C.; Weimer, W.; Zhang, F. Towards transparent introspection. In Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Osaka, Japan, 14–18 March 2016; Volume 1, pp. 248–259.
234. Wüchner, T.; Ochoa, M.; Pretschner, A. Robust and Effective Malware Detection Through Quantitative Data Flow Graph Metrics. In Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Milan, Italy, 9–10 July 2015; Volume 9148, pp. 98–118. [CrossRef]
235. Rhee, J.; Riley, R.; Xu, D.; Jiang, X. Defeating Dynamic Data Kernel Rootkit Attacks via VMM-Based Guest-Transparent Monitoring. In Proceedings of the 2009 International Conference on Availability, Reliability and Security, Jukuoka, Japan, 16–19 March 2009; pp. 74–81. [CrossRef]
236. Mutti, S.; Fratantonio, Y.; Bianchi, A.; Invernizzi, L.; Corbetta, J.; Kirat, D.; Kruegel, C.; Vigna, G. BareDroid: Large-Scale Analysis of Android Apps on Real Devices. In Proceedings of the ACSAC 2015, 31st Annual Computer Security Applications Conference, Angeles, CA, USA, 7–11 December 2015; pp. 71–80. [CrossRef]

237. Spensky, C.; Hu, H.; Leach, K. LO-PHI: Low-Observable Physical Host Instrumentation for Malware Analysis. In Proceedings of the Network and Distributed System Security Symposium, San Diego, CA, USA, 21–24 February 2016.
238. Zhou, L.; Xiao, J.; Leach, K.; Weimer, W.; Zhang, F.; Wang, G. Nighthawk: Transparent system introspection from ring-3. In Proceedings of the European Symposium on Research in Computer Security, Luxembourg, 23–27 September 2019; pp. 217–238.
239. Garfinkel, T.; Adams, K.; Warfield, A.; Franklin, J. Compatibility Is Not Transparency: VMM Detection Myths and Realities. In Proceedings of the USENIX Workshop on Hot Topics in Operating Systems, San Diego, CA, USA, 7–9 May 2007.
240. Besler, F.; Willems, C.; Hund, R. Countering innovative sandbox evasion techniques used by malware. In Proceedings of the 29th Annual FIRST Conference, San Juan, PR, USA, 11–16 June 2017.
241. Gajrani, J.; Sarswat, J.; Tripathi, M.; Laxmi, V.; Gaur, M.S.; Conti, M. A Robust Dynamic Analysis System Preventing SandBox Detection by Android Malware. In Proceedings of the SIN '15, 8th International Conference on Security of Information and Networks, Sousse, Tunisia, 11–13 November 2015; pp. 290–295. [CrossRef]
242. Hu, W.; Xiao, Z. Guess where i am-android: Detection and prevention of emulator evading on android. In Proceedings of the XFocus Information Security Conference (XCon), Beijing, China, 20–21 August 2014.
243. Dietze, C. Porting and Improving an Android Sandbox for Automated Assessment of Malware. Master's Thesis, Hochschule Darmstadt, Darmstadt, Germany, 2014.
244. D'Elia, D.C.; Invidia, L.; Palmaro, F.; Querzoni, L. Evaluating dynamic binary instrumentation systems for conspicuous features and artifacts. *Digit. Threat. Res. Pract.e (DTRAP)* **2022**, *3*, 1–13. [CrossRef]
245. Vidas, T.; Tan, J.; Nahata, J.; Tan, C.L.; Christin, N.; Tague, P. A5: Automated Analysis of Adversarial Android Applications. In Proceedings of the SPSM '14, 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices, Scottsdale, AZ, USA, 3–7 November 2014; pp. 39–50. [CrossRef]
246. Raffetseder, T.; Kruegel, C.; Kirda, E. Detecting System Emulators. In Proceedings of the Information Security, 10th International Conference, ISC 2007, Valparaiso, Chile, 9–12 October 2007; pp. 1–18.
247. Vidas, T.; Christin, N. Evading Android Runtime Analysis via Sandbox Detection. In Proceedings of the ASIA CCS '14, 9th ACM Symposium on Information, Computer and Communications Security, Kyoto, Japan, 4–6 June 2014; pp. 447–458. [CrossRef]
248. Yokoyama, A.; Ishii, K.; Tanabe, R.; Papa, Y.; Yoshioka, K.; Matsumoto, T.; Kasama, T.; Inoue, D.; Brengel, M.; Backes, M.; et al. Sandprint: Fingerprinting malware sandboxes to provide intelligence for sandbox evasion. In Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses, Paris, France, 19–21 September 2016; pp. 165–187.
249. Egele, M.; Scholte, T.; Kirda, E.; Kruegel, C. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv. (CSUR)* **2008**, *44*, 1–42. [CrossRef]
250. Yan, P.; Yan, Z. A survey on dynamic mobile malware detection. *Softw. Qual. J.* **2018**, *26*, 891–919. [CrossRef]
251. Sun, S.T.; Cuadros, A.; Beznosov, K. Android Rooting: Methods, Detection, and Evasion. In Proceedings of the SPSM '15, 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices, Denver, CO, USA, 2 October 2015; pp. 3–14. [CrossRef]
252. Alam, D.; Zaman, M.; Farah, T.; Rahman, R.; Hosain, M.S. Study of the dirty copy on write, a linux kernel memory allocation vulnerability. In Proceedings of the 2017 International Conference on Consumer Electronics and Devices (ICCED), London, UK, 14–14 July 2017; pp. 40–45.
253. Vidas, T.; Christin, N. Sweetening Android Lemon Markets: Measuring and Combating Malware in Application Marketplaces. In Proceedings of the CODASPY '13, Third ACM Conference on Data and Application Security and Privacy, San Antonio, TX, USA, 18–20 February 2013; pp. 197–208. [CrossRef]
254. Ashawa, M.; Morris, S. Analysis of mobile malware, evolution and infection strategies: A systematic review. *J. Inf. Secur. Cybercrimes Res.* **2021**, *4*, 1–29. [CrossRef]