Thèse de doctorat de

UNIVERSITÉ DE RENNES 1

L'UNIVERSITÉ DE RENNES 1

École Doctorale N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

Louison Gitzinger

Surviving the massive proliferation of mobile malware

Survivre à la prolifération massive des malwares sur mobile

Thèse présentée et soutenue à Rennes (France), le 8 décembre 2020 à 10h
Unité de recherche : Irisa (UMR 6074)

Rapporteurs avant soutenance :

Sonia BEN MOKHTAR	Directrice de recherche	CNRS Lyon
Romain ROUVOY	Professeur des universités	Université de Lille

Composition du Jury :

Rapporteurs :	Sonia BEN MOKHTAR	Directrice de recherche	CNRS Lyon
	Romain ROUVOY	Professeur des universités	Université de Lille
Examinateurs :	Tegawendé F. BISSYANDE	Directeur de recherche	Université du Luxembourg
	Alain TCHANA	Professeur des universités	ENS de Lyon
	Sophie PINCHINAT	Professeur des universités	Univ Rennes, CNRS, Inria, IRISA
	Gilles MULLER	Directeur de recherche	Inria
Dir. de thèse :	David-Yérom BROMBERG	Professeur des universités	Univ Rennes, CNRS, Inria, IRISA

Table of Contents

1 Introduction **7**

1.1 Context 7

1.2 Android ecosystem vulnerabilities 9

1.3 Existing solutions to fight malware profilferation in the ecosystem 10

1.4 Limits of malware defenses in the Android ecosystem 12

1.5 Thesis statement - Contributions 14

2 State of the Art **17**

2.1 Hunting threats in the Android ecosystem 17

2.1.1 Improving privacy on Android 17

2.1.2 Reducing the attack space on the Android platform 21

2.2 Android Malware detection 24

2.2.1 Signature-based malware detection. 25

2.2.2 Machine learning-based detection. 26

2.3 Adversarial Attacks to Malware Detection 30

2.3.1 Review of existing studies 31

2.3.2 Open problems 35

2.4 Defenses against adversarial attacks 37

2.5 Conclusion 39

3 Exploring malware detection tools on Android **41**

3.1 Background on the Android ecosystem 41

3.1.1 Android platform architecture 41

3.1.2 Android application architecture 42

3.1.3 Malware infection in the Android ecosystem 44

3.2 Program analysis 46

3.2.1 Limits of dynamic analysis to malware detection 46

3.2.2 Advantages of static analysis in malware detection 47

3.3 Techniques to perform malware detection at scale 48

3.3.1 Signature based detection . . . 48
3.3.2 Machine learning assisted detection . . . 49
3.4 Improving existing malware detection systems . . . 50
3.4.1 DroidAutoML: hyper-parameter optimization for ML scanners . 50
3.4.2 DroidAutoML approach . . . 52
3.4.3 DroidAutoML evaluation . . . 56
3.4.4 Groom: A feature engineering framework to reach more efficient ML malware detection models . . . 60
3.4.5 Groom approach . . . 63
3.4.6 Groom evaluation . . . 66
3.5 Conclusion . . . 68

4 Reaching limits of antivirus leveraging the creation of experimental datasets 71
4.1 Evolution of Android malware datasets . . . 71
4.1.1 Obfuscation of Android applications. . . . 72
4.1.2 Review of recent evasion techniques on Android . . . 72
4.2 KillerDroid: Weaving Malicious Payloads with Benign Carriers to Massively Diversify Experimental Malware Datasets . . . 77
4.2.1 Approach . . . 78
4.2.2 Experimental Setup . . . 85
4.2.3 Evaluation . . . 89
4.3 Conclusion . . . 105

5 Increasing the quality of ground truth datasets on Android 107
5.1 Limits of antivirus aggregators solutions to build ground truth datasets . . 108
5.2 KillerTotal: Vetting grand public Antivirus products . . . 110
5.2.1 Mobile Antivirus products . . . 110
5.2.2 Approach . . . 111
5.2.3 Evaluation . . . 114
5.3 KillerScreenshot: Improving the quality of malware datasets . . . 119
5.3.1 Approach . . . 119
5.3.2 Evaluation . . . 122
5.4 Conclusion . . . 123

6 Future works **125**
6.1 Short term future works . . . 125
6.1.1 Towards more diversified malware datasets . . . 125
6.1.2 Future of machine learning malware detection . . . 128
6.1.3 Poisoning VirusTotal . . . 130
6.2 Long term future works . . . 132
6.2.1 Towards more collaborative efforts . . . 132
6.2.2 Moving applications to the cloud . . . 132

Conclusion **135**
6.3 Summary of contributions . . . 135

7 Résumé en français **141**
7.1 Contexte . . . 141
7.2 Vulnérabilité de l'écosystème Android . . . 143
7.3 Solutions existantes . . . 143
7.4 Limites des solutions existantes dans l'écosystème . . . 145
7.5 Contribution de la thèse . . . 147

Bibliography **149**

Chapter 1

Introduction

1.1 Context

Nowadays, many of us are surrounded by a fleet of electronic devices, connected to each other using different network protocols. These devices are often qualified as *smart*: smartphones, smart TVs or smart watches, as they seamlessly operate interactively and autonomously together with multiple services to make our lives more comfortable. For example, your smartphone can notify you at work that your smart camera detected your children are back home. When planning a road trip with the help of your smartphone, your smart car will seamlessly know how to guide you to your final destination. To make such scenarii possible, smart devices we use every day are part of larger ecosystems, in which various companies collaborate with each other. From a business perspective, these ecosystems can be seen as an inter-dependence between supply (companies selling devices and services) and demand (users). To stay profitable over time and maximize user retention, companies gain users' trust by guaranteeing their security and offering them a good quality of service.

To offer strong and coherent ecosystems, companies, like Google and Apple, have organised their business around a software platform, such as Android or IOS. Such platforms (see figure 1.1, ❶) power smart devices (see figure 1.1, ❷) and allow users to install programs called *applications* (see figure 1.1, ❸). Applications are individual software units that offer users various services for productivity (sending SMSs and emails, managing agenda, ...), entertainment (social medias, videos, ...) or even gaming. Software platforms come with a Software Development Kit (SDK) which encourages and facilitates the development of new applications compatible with the overall platform inside the ecosystem. SDKs open up the ecosystem to the developer community (see figure 1.1, ❹) and allow any third-party company (Airbnb, Facebook, Uber, ...) to design applications for smart device users.

Once installed, applications can leverage the many device's sensors (camera, fingerprint

scanner, accelerometer, GPS location, . . .) to provide additional functionalities to the user, such as taking pictures, fingerprint authentication, navigation, etc. With the help of device communication means (LTE, WiFi, Bluetooth, . . .), applications can further send acquired data to third-party servers, enabling real time feedback loops, providing developers useful quality feedbacks on their users. In exchange for these data, users can enjoy additional features in their applications, such as tailored advertisement, faster ordering, customized services, or easy payment in just one click.

To commercialize applications created by developers, online platforms, called *stores* (see figure 1.1, ❺), offer users a centralized place to browse and download them. As digital marketplaces, stores provide mechanisms to vendors (i.e. developers) to upload, publish and advertise their applications on the service. To help users navigate the millions of applications published, stores organize them into categories, such as *Art & Design*, *Entertainment* or *Food & Drink*.

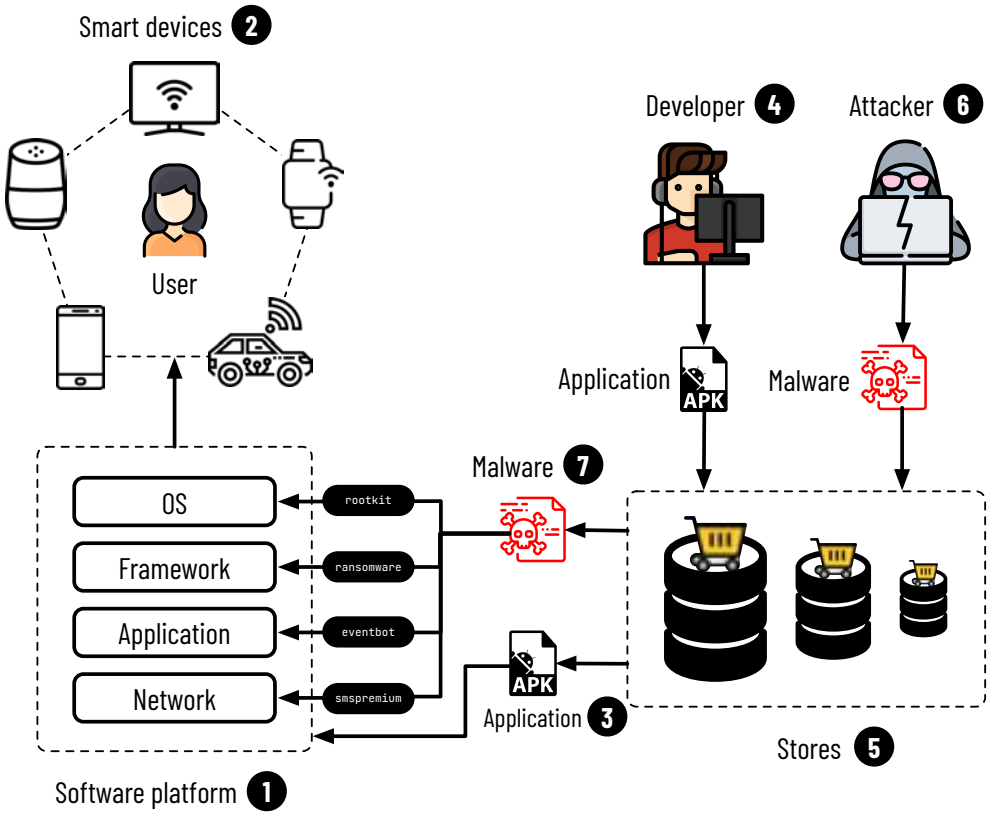

Figure 1.1 – Overview of a smart device ecosystem

Over time, these smart device ecosystems thrived, and became profitable for all stake-

holders. This virtuous circle between all actors of the ecosystem helped dramatically increase the business of smart devices. Software platforms account for billions of active devices around the world, led by Android and Apple IOS which together hold 98% of the market [1]. They respectively run 2.5 and 1.4 billions of devices [2, 3]. Following this trend, application stores have naturally seen a tremendous increase in activity. In ten years, the number of published applications for the two leading stores (namely the Google Play Store and the Apple App Store) went from a few thousand to several million [4, 5] (from 38 000 to 3 millions for Google and 218 000 to 4.3 millions for Apple respectively).

To stay profitable in this context, ecosystems must maintain the user's trust. To do so, stores must ensure that applications published on their platforms are of good quality, and do not have bad intentions towards users. However, the growth of these ecosystems has led bad actors (e.g. attackers, see figure 1.1, ❻) to take advantage of the system in an illegitimate manner.

1.2 Android ecosystem vulnerabilities

Malicious people may leverage vulnerabilities at all levels of the ecosystem to target users with malicious applications (see figure 1.1, ❼), namely malware, that stealthily exploit users' data and device vulnerabilities to spy on them or make money off of them. In this context, due to its popularity, the Android ecosystem is targeted by 98% of attacks on smart devices [6]. To exploit vulnerabilities in the Android ecosystem, attackers create many kinds of malware, thereafter categorized into families [7]. These malware applications are specifically designed to exploit vulnerabilities at several levels of the software platform to attack smart device users.

— At the operating system level (see figure 1.1, `rootkit`): attackers can exploit vulnerabilities to deploy *rootkits*, to gain *root* access on the target device. In 2019 the `xHelper` malware infected at least 45 000 devices and obtained root on them by remotely fetching a malicious payload [8].

— At the framework level (see figure 1.1, `ransomware`): malware can be granted access to potentially sensitive users' data such as their GPS location or their contact list. Malware can also ask the framework permissions to perform dangerous actions such as deleting users' files, taking pictures or displaying sticky overlays. For example, *Ransomware* encrypt the users' filesystem and block the user interface until users aggree to pay a ransom to obtain the decryption key to recover their files [9].

— At the application level (see figure 1.1, `eventbot`): malware can exploit other vulnerable applications, for example, to stop the faulty application's activity or access sensitive information. The `EventBot` malware, discovered in 2020, exploit banking applications to steal users' financial data [10].

— At the network level (see figure 1.1, `smspremium`): malware can perform deny of services on other applications, send SMS or make phone calls. In 2017, Google removed 50 malware applications from its store that were stealthily sending SMSs to premium numbers in the background [11].

The worrisome growth of malware threats in the Android ecosystem presents a daunting challenge: how to keep an ecosystem open to third-party contributors while maintaining security guarantees for users?

1.3 Existing solutions to fight malware profilferation in the ecosystem

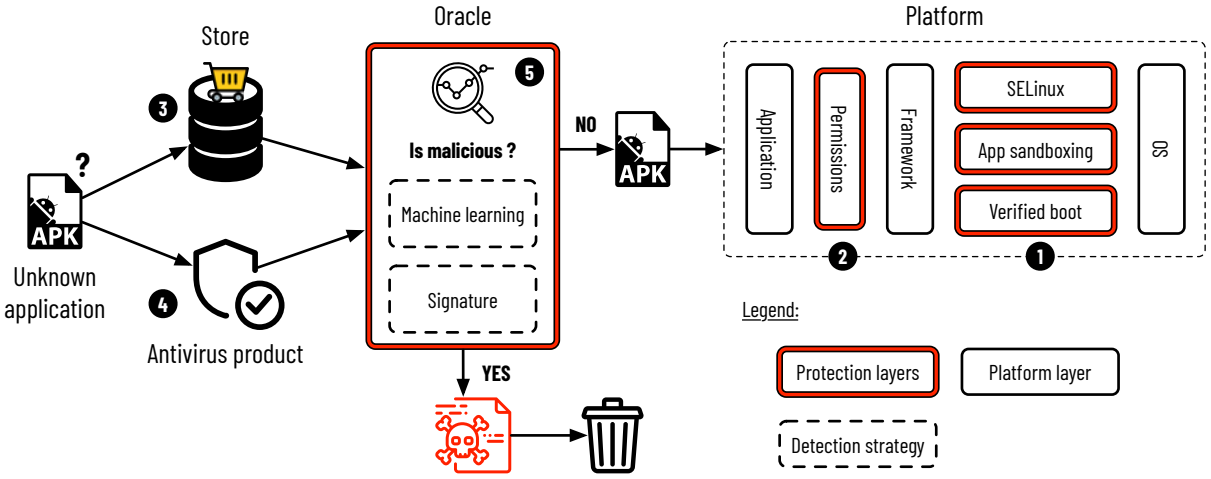

Figure 1.2 – Overview of Android ecosystem defenses against malware threats

To thwart the malware threat, Android security actors, from both research and industry, continuously reinforce their security level with new defenses. The Android platform leverages its kernel (i.e. Linux) capabilities (see figure 1.2, ❶), such as application sandboxing [12], SELinux [13] and verified boot [14] to better isolate running applications from each other and control access to application data. With its permission model (see figure 1.2, ❷), which has been subject to many improvements in recent years [15], the

Android framework explicitly asks applications to define the capabilities they need and will deny any attempt to access unauthorized resources. Therefore, to access functions such as camera, telephony, sms/mms and network, an application must explicitly ask permission from the user. These platform-level defenses provide a first level of security to prevent malicious applications from doing too much damage when they manage to infect a device. However, the growing number of mobile attacks [16] show that these defenses are not sufficient to stop malware infections.

To slow down the infection of devices by malicious applications upstream, stores (see figure 1.2, ❸) have put in place validation systems to filter applications submitted on their platform. These validation processes, employ both manual [17] and automated [18] techniques to analyse the stream of uploaded applications in order to determine their legitimacy before publishing them. These protections are not sufficient to suppress every malware, which motivates the need for on-device antivirus (see figure 1.2, ❹) that detect malware threats when a new application is installed. In addition, the research domain has shown promising results with novels approaches to improve Android malware detection [19–22].

Establishing the legitimacy of an application requires analysing it to collect insightful data that can provide information on its intention and behaviour. To do so, two software analysis techniques are mainly used: (i) static program analysis, which allow for a quick analysis of application binaries and resource files without needing to execute it, and (ii) dynamic analysis, which consists in executing the application with multiple input values to analyse its runtime behaviour.

Afterwards, to automatically decide whether an unknown application should be considered legitimate or malicious, detection systems use an oracle (see figure 1.2, ❺). Oracles are decision systems that make a choice, i.e. label an application as legitimate or malicious, based on the data collected during application analysis. To operate, oracles mainly leverage two different strategies:

— Signature-based detection: with signature-based detection, oracles make its decision by comparing information such as file signature or bytecode patterns with sets of heuristics and rules generated from previously seen malware.
— Machine learning-based detection: with machine learning-based detection, oracles use existing malware databases to train classifiers that predict the maliciousness of an application.

1.4 Limits of malware defenses in the Android ecosystem

Despite all the solutions put in place, a recent study [23] has shown that at least 10 000 malware belonging to 56 malware families were successfully published on the Google Play Store between 2014 and 2018. Moreover, security reports from security companies [6, 16, 24] show that the number of Android devices infected by malware is still on the rise. In fact, attackers become smarter over time and use more and more advanced techniques to defeat security defenses put in place by Android ecosystem actors.

To defeat static analysis, attackers use complex evasion techniques such as *dynamic code loading* [25], *reflection* [26] or *encryption* [27] to hide their malicious payload within the application and make it unreachable to static analysis programs. Regarding dynamic analysis, detection systems often execute applications within virtual machines instead of real devices to analyse applications at scale. However, Petsas et al. [28] have shown that malware can detect when they are running in a virtual environment. To defeat signature-based detection, attackers use polymorphism [27] and obfuscation [29] that allow them to completely change the structure of the application's code without affecting its semantics, to rapidly craft malware variants with unforeseen signatures. In an attempt to counter the above phenomenon, machine learning (ML) is massively adopted to improve malware detection accuracy [19–21, 30]. However, current advances in ML-based malware detection on Android may suffer from several limitations that we summarize in 3 distinct claims.

Claim 1: limits of existing machine learning solutions. Although it is a first step toward improving detection, most of machine learning related studies neglect the search for fine-tuned learning algorithms. Performances of trained machine learning classifiers depend strongly on several key aspects: (i) learning algorithms, (ii) parameters (i.e. hyper-parameters) and (iii) feature engineering. As far as our knowledge goes, few attempts have been made towards optimizing existing Android ML-based detection approaches. Whether one [20, 21] or several algorithms [19, 31, 32] are evaluated, the evaluations are always carried out empirically, implying a manual process with few and/or default hyper-parameter combinations. Besides, features extracted by recently proposed approaches [19, 21] neglect new evasion techniques used by malware such as dynamic code loading or reflection, that allow attackers to dissimulate the malware's behaviour.

Claim 2: limits of publicly available malware datasets. Regarding machine learning based detection, recent studies [33, 34] have shown that specifically crafted malware, namely adversarial examples, can successfully evade machine learning classification models. These studies emphasise the lack of heterogeneous malware datasets that include all kinds of malware samples to train ML-based detection systems. To train a reliable classifier, malware datasets need to include a large variety of samples that allow machine learning model to better generalize the problem and catch corner case samples. Specifically, existing malware datasets lack malware samples that use complex evasion techniques such as dynamic code loading, encryption and reflection.

Claim 3: limits of malware samples verification methods. A fundamental step towards implementing and evaluating novel malware detection methods is the creation of high quality ground truth datasets used to train those methods. The efficiency of a malware detection method is directly correlated [35, 36] with the quality of the dataset used to train and test it. However, the quality of a ground truth dataset can be impeded by (i) inacurrate sample labelling [36], i.e. legitimate samples labeled as malicious or *vice versa*, and (ii) unrealistic samples, i.e. outdated or non-working samples.

The vast majority [21, 22, 37–41] of ML-based malware detection approaches rely on antivirus aggregators to determine whether an application is legitimate or malicious and further build training datasets. Antivirus aggregators (such as VirusTotal [42]) are online platforms that run many antivirus products from third-party security companies to vet files uploaded by users. However, it has been suggested [23, 43] that such antivirus aggregators may introduce bias in datasets built upon this selection method, due to erroneous sample scan results. In fact, for commercial or technical reasons, security companies may provide aggregators with a restrained version of their antivirus engine that may artificially lower the accuracy of aggregators compared to what could be possible with fully-fledged engines.

Current malware detection approaches in the Android ecosystem struggle to deal with ever smarter ill-intentioned people. On one side, traditional signature-based approaches can not efficiently detect new malware variants generated at a high paced rate by attackers. On the other side, ML-based approaches are facing problems inherent to the machine learning domain: they need to be constantly updated and optimized to keep detecting new malware samples, and the lack of quality malware datasets negatively impacts the

quality of trained classifiers.

1.5 Thesis statement - Contributions

To address the aforementioned problems, additional efforts are required. Therefore, we contribute to the research community efforts with the following contributions:

Automating the evaluation of ML detection systems. First off, to address **Claim 1**, we propose DroidAutoML, a novel approach to automating the evaluation of machine learning based malware detection systems on Android. DroidAutoML addresses the problem of ML algorithm selection and hyper-parameter optimization by leveraging *AutoML* to improve the accuracy of ML-based malware detection systems. DroidAutoML is further built on a dedicated microservice architecture specifically designed to fulfill the need for genericity and flexibility required by the Android malware detection domain.

Novel set of features to improve ML detection. As our second contribution to address **Claim 1**, we propose Groom, to address the problem of feature engineering in the context of machine learning based malware detection on Android. Groom is a framework that leverages a broad static analysis to drastically improve the quality of features extracted from an application, while being faster than existing approaches. In particular, Groom extracts specific features that enable detection systems to take into account the use of recent obfuscation techniques such as reflection, native code execution, dynamic code loading and encryption.

Toward massively diversifying experimental malware datasets. To address **Claim 2**, we propose KillerDroid, a toolchain that enables to craft new malware variants that use advanced evasion techniques with the aim of creating samples to massively diversify experimental malware datasets. KillerDroid can generate malware variants at scale to evaluate the robustness of state-of-the art malware scanners, both from academia and the industry, against adversarial attacks.

Increasing the quality of ground truth datasets on Android. Finally, to address **Claim 3**, we propose two approaches to improve the overall quality of experimental malware datasets used to train and test malware detection systems.

We first propose KillerTotal, a novel approach for evaluating individually the publicly available version of the seven most efficient Android antivirus products available on the Google Play Store. KillerTotal aims at showing that mobile versions of commercial antivirus are more accurate and more robust to detect malware samples than engines hosted in antivirus aggregators such as VirusTotal [42]. Our system provides a large-scale deployment platform based on Android Emulators to automate the initialisation and execution of mobile antivirus products.

Secondly, we propose KillerScreenshot, a framework that automatically certifies that malware variants synthetically produced by adversarial production toolchains (such as KillerDroid) are fully functional. KillerScreenshot dynamically verifies that the behaviour of a crafted malware variant is consistent with the behaviour of the original malware used to create it by comparing the screen activity of both samples.

Dissertation outline

The remainder of this thesis is structured as follows. In chapter 3, we begin by providing background regarding the Android platform. Then, we discuss several limits of existing machine learning-based malware detection solutions in the Android ecosystem. Finally, we present and evaluate our approaches: DroidAutoML and Groom. In chapter 4, we discuss the unreliability of public existing malware datasets in the Android research community. We further provide an extended analysis regarding new evasion techniques used by malware authors to defeat scanners. Afterwards, we present and evaluate KillerDroid, our approach to generating new malware variants to improve the overall quality of existing datasets and challenge scanners. In chapter 5, we discuss the limits of restrained antivirus aggregators and present our alternative, KillerTotal, based on fully-fledged mobile antivirus products sold on the Google Play Store. We further present scalable approaches to verify the correct behaviour of automatically crafted malware variants.

Chapter 2

State of the Art

This chapter aims at reviewing the state of research in domains that are closely related to the problem we are addressing: to globally improve the security of the Android ecosystem by tackling the invasion of malicious software. First off, we review existing solutions to mitigate existing vulnerabilities on the Android platform, especially regarding user's privacy. Secondly we review (i) studies related to Android malware detection, and (ii) studies specifically addressing the problem of adversarial attacks against malware detection scanners.

2.1 Hunting threats in the Android ecosystem

More and more privacy-sensitive data are processed on smart devices, namely data that relate to a person that is not necessarily willing to share. Our devices become a central location vulnerable to many external threats, such as data leakage or erasure, third-party surveillance or identity theft. Consequently, securing smart devices involves a meticulous search of unknown vulnerabilities on which attackers can rely to carry out their attacks. In addition to security layers put in place on Android (such as *SELinux*, application sandboxing, the permission level, etc.) many research works [44–48] proposed solutions to globally raise the security level of the Android platform. This section presents the most prevalent approaches and solution for tracking vulnerabilities on the Android system.

2.1.1 Improving privacy on Android

In the last years, many efforts have been observed to improve the privacy of Android users, particularly by preventing applications from illegitimately accessing user's sensitive data. As such, Android platform developers and maintainers put a lot of efforts to improve the Android permission model [15] to fight against abusively privileged applications. The

aim of the permission model is to protect the privacy of an Android user by forcing applications to request permission to access sensitive data (such as contacts and SMS) or system features (such as camera or internet).

However, recent studies [45, 49, 50] shown that these improvements are not sufficient to prevent applications from performing dangerous actions on the device. With WOODPECKER [50], authors perform a data-flow analysis on running applications to demonstrate that eight popular Android stock images (i.e. different versions of the Android operating system) do not properly enforce the Android permission model, thus allowing applications to access user's sensitive data without requesting the related permission. With AXPLORER [45], authors show that permission checks in the Android Framework API internals are sometimes inconsistent, thus allowing applications to call unprotected API methods to access sensitive resources. In 2019, Reardon et al. [49], analysed the network traffic of hundreds of applications from the Google Play Store to detect sensitive data being sent over the network for which the sending application did not have the permission to access it. By reverse engineering such applications, authors discovered several side channels in the framework that allow applications to access sensitive data without having the permission to do so.

Revealing privacy leaks with taint tracking

Study	Year	Platform level	Objective	Approach
Lu et al. [47] (CHEX)	2012	Application	Detect component hijacking vulnerabilities in applications	static analysis
Artz et al. [51] (FLOWDROID)	2014	Application	Detect privacy leaks in applications	static analysis with taint-tracking
Enck et al. [51] (TAINTDROID)	2014	Application	Detect privacy leaks in applications	dynamic analysis with taint-tracking

Table 2.1 – Overview of studies that aim at detecting privacy leaks on the Android platform

To solve the problem of sensitive data leakage, several works [47, 51–53] (see table 2.1) proposed approaches to track potential data leakage in Android applications while taking into account subtleties of the Android framework such as the lifecycle of an application, or the permission model. With CHEX [47], authors propose a static analysis method to automatically detect if an application can be hijacked by another and therefore leak sensitive data. In 2014, Artz et al. [51] propose FLOWDROID, a static taint-analysis framework

that analyses the application's bytecode and configuration files to find potential privacy leaks, from a source (i.e. where data is collected) to a sink (i.e. where data flows out of the application). The FlowDroid framework helped to uncover sensitive data leaks in more than 500 applications from the Google Play Store and 1000 malware applications. With TaintDroid, authors propose an approach based on a dynamic analysis to monitor how applications handle user's data in realtime. More precisely, they use a *taint tracking* technique to monitor how sensitive information can flow out of the application context. TaintDroid allowed to demonstrate that 15 popular applications were actively sending device's location to third-party servers. More recently, Exodus [54], proposed a static analysis tool to determine whether an application embeds *trackers*, i.e. third-party libraries meant to collect data about user's usages. Exodus found that *Facebook analytics* is present in 17% of applications on the Google Play Store, *Google Analytics* on 25% and *Admob* on 67%.

Lessons learned. The aforementioned studies provide evidences to the research community that the leak of user's privacy sensitive data on the Android platform is a reality, whether for legitimate or illegitimate reasons. In particular, they show that an attacker can leverage a broad set of permissions genuinely granted by the user to leak sensitive information such as user's contacts, GPS location or SMS messages.

Detecting privacy leaks in inter application communication.

Study	Year	Platform level	Objective	Approach
Alhanahnah et al. [55] (DINA)	2019	Framework	Detect vulnerable inter-app communication in dynamically loaded code	static and dynamic analysis
Lee et al. [56] (Sealant)	2017	Application	Detect and visualize inter-app communication vulnerabilities in applications	static analysis
Octeau et al. [57] (Epicc)	2013	Application	Find and map inter-component communication in applications	static analysis
Bartel et al. [53] (IccTA)	2015	Application	Detect leaks among components in Android applications	static analysis
Hay et al. [58] (IntentDroid)	2015	Framework	Dynamic detection of inter-app communication	dynamic analysis

Table 2.2 – Overview of studies that aim at uncovering vulnerabilities related to Inter-Component Communication

Another source of privacy leaks on the Android platform is the communication mechanism between installed applications and the underlying framework. Android uses a messaging system that allow applications to communicate with the system or with each other through *Intents* [59]. Android components (i.e. pieces of software that make up an application) use *Intent* to communicate with each other (Inter-Component Communication, ICC), or across applications. As such, third-party applications can use *Intents* to shutdown the system, read the device internal storage or send any kind of data to another application. While the Android framework already provides a security mechanism to strengthen *Intent* security, many studies (see table 2.2) have discovered related vulnerabilities and proposed approaches to mitigate them.

In 2013, Backes et al. [57] propose a static analysis framework to analyse applications in the aim of uncovering vulnerable ICC method calls, i.e. that allow to leak sensitive uers's data. They found more than 2000 ICC leaks in 15 000 applications from the Google Play Store. With IntentDroid [58], authors instrument the Android framework with a custom dynamic analysis that allow to monitor communication between installed application at runtime. The approach allowed to reveal 150 privacy related vulnerabilities in 80 top applications from the Google Play Store. In 2015, Li et al. proposed IccTA [53], a static analysis framework that use a context aware data-flow analysis to detect privacy leaks between applications, especially between components of different applications installed on a device. IccTA allowed to discover privacy leaks in 337 application among 15 000 from the Google Play Store. With Sealant [56], authors proposed a static analysis approach intended for developers which allows to detect and visualize potentially vulnerable ICC within an application. More recently, Alhanahnah et al. proposed DINA [55], a static and dynamic analysis framework that detect vulnerable inter-application communication in dynamically loaded bytecode, thus allowing to uncover new information leakage vulnerabilities in dissimulated bytecode.

Lessons learned. These studies highlight the fact that security locks put in place by the Android framework are not invulnerable. Framework properties such as application-to-system or inter-application interactions can be hijacked by illegitimate applications to leak user's privacy sensitive data. The Android framework faces a trade-off between offering features to enhance third-party applications and guaranteeing a sufficient security level to users.

While these studies are useful to uncover vulnerabilities in Android applications, they do not provide mechanisms to actually protect the user. Such studies constitute an important step towards protecting the user from potential threats, especially regarding the user's privacy, but they are not sufficient to counter complex attacks specifically designed to cause damage. More importantly, the aforementioned approaches cannot determine if an application is manipulating user's sensitive data for a legitimate purpose or not. Therefore, these approaches cannot identify whether the behaviour of an application is legitimate or fraudulent. In numerous cases, legitimate applications require user's data to operate correctly, thus the final choice of keeping or discarding the suspicious application must be left to the user, thus leading to misinterpretation.

2.1.2 Reducing the attack space on the Android platform

Study	Year	Platform level	Objective	Approach
Rasthofer et al. [44] (DroidForce)	2016	Application	Reinforce permission model	static analysis and application instrumentation
Bugiel et al. [60]	2013	Kernel and Framework	Enforce mandatory access control	Implementation of a system-wide security architecture
Jing et al. [61] (IntentScope)	2016	Framework	Secure inter-app communication	Framework security extension
Zhang et al. [62] (AppSealer)	2014	Application	Detect and patch inter-app communication in applications	static and dynamic analysis
Backes et al. [63] (Boxify)	2015	Application	Better application isolation	application virtualization
Bianchi et al. [64] (NJAS)	2015	Application	Better application isolation	application virtualization using system call interposition
Yu et al. [65]	2013	Application	Secure Webview	Instrument application to protect sensitive call and ask user to accept or deny a specific call
Tuncay et al. [66] (Draco)	2016	Application	Secure Webview	Provide the developer a DSL to specify access control policy on resources exposed to Webview

Table 2.3 – Overview of studies that aim at reducing the attack space on the Android platform

Vulnerabilities in the Android platform are regularly discovered [7, 49, 67–69]. To mitigate them and reduce the attack space as much as possible, the Android research community have proposed many approaches (see table 2.3) to address the problem.

Reinforcing control access to sensitive resources.

To reinforce and control access to sensitive resources, studies [44, 60] explored techniques that allow the user to have a fine-grained control over the flow of data entering and leaving its device. In 2013, Bugiel et al. [60] proposed a *system-wide* security architecture for the Android operating system to enforce resource access control with a custom security policy both at the kernel and the framework level. The study is at the origin of the latter integration of SELinux in the Android Open Source Project (AOSP) [70] and allowed to mitigate root exploits. With DroidForce [44], authors propose a framework, implemented at the application level, that allow to instrument an Android application to inject custom checks to enforce sensitive data policies enforced by the user. DroidForce provides a custom domain specific language that allow the user to set up privacy aware policies such as *"This application is allowed to send 2 SMSs per day maximum"*.

Lessons learned. Aforementioned studies restrain access to sensitive resources on the device while keeping the user in control to allow legitimate applications to do so. Such studies demonstrate that new security methods can be implemented at several layers of the Android platform to keep the user in control of its data. However, similarly to the Android permission model, such security approaches still rely on the capacity of the user to correctly interpret the intentions of an application and decide whether to trust it or not.

Application sandboxing

The Android platform leverages on Linux user-based protection to isolate application resources. In the Linux system, each user is assigned a unique user id (UID), and users are isolated by the kernel in such a way that one user cannot access the data of another user. On Android, the system assigns a unique user id to each Android application and runs it in its own process. This mechanism allows to isolate applications from each other and protects the system from potential malicious behaviours. The Android sandbox is in the kernel, thus if one malicious application tries to read the data of another application, this is not permitted as the malicious application does not have user privileges. Over years, several works [7, 8, 67] demonstrated that the application sandbox model is not invulnerable, which drove the search for a better isolation between running applications. With Boxify, Backes et al. [63] propose an *app sandoxing* concept which allows to encapsulate untrusted applications within a controlled and trusted environment without modifying the

application framework. In particular, the mechanism provides a better application isolation by revoking by default the permissions for I/0 operations to untrusted applications. Similarly, Bianchi et al. proposed NJAS [64], another *app virtualization* approach that use the `ptrace` mechanism to perform syscall interposition. Both approaches use auto-generated generic *stub* applications that serve as a container to sandbox the untrusted application.

Lessons learned. These sandboxing methods are hard to maintain because they require a lot of engineering work to keep the sandbox code compatible with the underlying framework. These sandboxing mechanisms often rely on Android framework features that can be protected or removed in future releases of the platform. For example, since Android 8.0, it is not possible anymore for an Android application to use the `ptrace` tool, thus making the NJAS approach unusable. Moreover, a malicious application can be specifically crafted to gather information about its runtime environment and change its runtime behaviour when being sandboxed. For example, a malware could detect that the application's threads are monitored by `ptrace` when sandboxed by the NJAS approach. To the best of our knowledge, there is currently no more efficient way than the built-in Android application sandboxing to fully isolate an application on the platform.

Securing inter-component communication

To protect user from vulnerabilities related to Inter-Component Communication, Jing et al. [61] proposed IntentScore a static analysis tool along with a security extension to the Android platform to detect vulnerable ICC methods and propose developers a patch to prevent information leaks between applications. IntentScope has proven to be able to detect applications vulnerable to eavesdropping as well as applications that communicates user's privacy-sensitive data to under-priviledged applications. Similarly, authors of AppSealer [62] proposed a static and dynamic analysis framework to detect vulnerable ICC methods in an application and instrument it to patch found vulnerabilities.

Lesson learned. These studies propose an additional step towards securing the Android platform by providing application developers insightful tools to develop less vulnerable applications. However, to avoid hampering legitimate applications, proposed approaches cannot afford to automatically patch any vulnerability found on applications, as they are not able to know whether scanned applications are legitimate or not.

Securing hybrid applications

Hybrid applications are applications designed using the *Webkit* technology and developed with Javascript, HTML and CSS. Hybrid applications are more and more used because they allow developers to build the applications for several platforms (Android, IOS, Web) by re-using the same codebase. The emergence of hybrid applications raised concern about web vulnerabilities on Android [71, 72]. Several studies [69, 72] shown that well known web attacks such as cross-site scripting are applicable on Android within the *Webview* context.

Following the discovery of these vulnerabilites, the version 4.3 of Android patched several *Webview* related vulnerabilities and several research works [65, 66] proposed approaches to mitigate such attacks. In 2013, Yu et al. [65] proposed a static analysis framework to instrument applications that use Webview to enforce a security policy on sensitive API methods called from the context of webview. Later, authors of Draco [66] designed a domain specific language (DSL) to allow developers to specify specific access control policies on resources exposed to a Webview context.

All the aforementioned studies have made it possible to greatly strengthen the Android platform over time. However, these new defenses put in place must be done while leaving the possibility for legitimate applications to take advantage of system's features to properly serve the user. Actually, nothing prevents an attacker from building a malicious application that masquerades as legitimate for user's eyes and stealthily use systems features to make damages on the device or leak user's sensitive data. This makes it particularly difficult to distinguish a legitimate application from a malware by simply comparing their behaviors. In this thesis, we take a closer look at systems that try to detect and discard malicious applications while keeping legitimate applications within the ecosystem.

2.2 Android Malware detection

Despite the efforts of the Android Open Source Project (AOSP) to secure the platform, the Android ecosystem is, since its emergence on the market, massively attacked [73]. This rise in cybercrime highlights the need for more effective measures to control the infection rate of smart devices. Consequently, numerous studies started to address the problem of malware detection on Android.

2.2.1 Signature-based malware detection.

Study	Year	Application analysis type	Features
DroidMOSS [74]	2012	static analysis	Dalvik bytecode (opcodes)
Zhou et al. [75]	2013	static analysis	API calls, permissions
Kolbitsch et al. [76]	2009	dynamic analysis	system calls

Table 2.4 – Overview of studies leveraging a signature-based approach for malware detection

As a first strategy for malware detection, several studies (see table 2.4) and commercially available Android antivirus rely on signature-based detection approaches. Signature-based detection consists in collecting, by static or dynamic analysis, a signature from an unknown application and checking whether this signature corresponds to an identified malware in a database kept up to date. Hence, this technique is fast to detect already seen malware but leaves users exposed to new unseen malware until the signature is available in the underlying database.

In 2009, Kolbitsch et al. [76] proposed a malware detection approach based on dynamic analysis that produces a behaviour graph of the analyzed software by monitoring its system calls. Then the approach looks for similar existing graphs in a database to determine the analyzed software is malicious or not. On mobile devices, several studies [75, 77] proposed frameworks that use application signature to detect repackaged applications in Android marketplaces (i.e. stores). With DroidMOSS [74], authors used static analysis to extract opcodes from the application bytecode and compute a signature using a fuzzy hashing technique. They demonstrated that 5 to 13% of applications on third-party Android marketplaces are repackaged. Zhou et al. [75] proposed an alternative approach by computing application signature using framework API calls and application's permission. Authors demonstrated that 1% of applications on the official Android marketplace (currently named Google Play Store) are repackaged.

However, it has been shown that attackers can evade signature-based detection by quickly generating new malware variants. For example, the use of polymorphism [78] and obfuscation [79] allows to completely change the structure of the code of an application without affecting its semantics, thus allowing attackers to quickly craft new malware variants with unforeseen signatures.

2.2.2 Machine learning-based detection.

Study	Year	Learning algorithm	Feature extraction	Features
Drebin [21]	2014	SVM	static analysis	permissions, suspicious API calls, protected API calls, url strings in bytecode
MaMaDroid [19]	2017	SVM, RF, KNN	static analysis	Abstracted sequence of API calls encoded with Markov Chains
DroidAPIMiner [20]	2013	DT, KNN, SVM	static analysis	8375 framework API calls
Peiravian et al. [80]	2013	SVM, DT, bagging predictors	static analysis	1326 API calls and 130 permissions
Andromaly [32]	2012	K-means, logistic regression, DT, Bayesian networks, Naïve Bayes	dynamic analysis	CPU load, memory usage, power, Network traffic
MADAM [81]	2018	KNN	dynamic analysis	system calls, user activity, network traffic
MARVIN [82]	2015	Linear classifier, SVM	static and dynamic analysis	App names, syscalls, network traffic, crypto operation, manifest metadata, phone activity, permissions

Table 2.5 – Overview of studies that aim at detecting malware in the Android ecosystem

For these reasons, new malware detection approaches based machine learning (ML) have emerged [19–21, 30, 80]. ML-based malware detection consists in training a statistical model using a learning algorithm (such as Random Forest or Support Vector Machine) able to reason on properties learned from previously seen data to predict the maliciousness of an unforeseen sample. To perform malware detection, aforementioned studies are based on supervised machine learning tasks, that said learning a function that maps inputs to outputs based on previously seen input-output pairs. In the last decade, several studies (see table 2.5) proposed approaches that achieve good scores and reached over 90% accuracy when classifying malware and benign applications with a very low false positive rate.

However, as we will see in the next section and throughout this thesis, accuracy of machine-learning based detection systems depends on the experimental setup used to evaluate them. In particular, the capacity of these approaches to efficiently distinguish benign applications from malicious ones depend on (i) the information (i.e. features) extracted from applications, (ii) the learning algorithm used to train the model, and (iii) the quality of the training dataset. In this thesis, we will see that an open problem in the field is to verify that these results can be verified in other experimental setups than those proposed in original studies.

Review of existing approaches

In machine learning, a studied object is often described by features, i.e. individual and measurable properties of the object. To be able to classify an application as benign or malicious, a machine learning-based approach relies on a learning algorithm that require a numerical representation of applications, namely feature vectors, given as input to facilitate processing. A feature vector is a n-dimensional vector of features that represent an object. In this context, *feature engineering* is the action of choosing, extracting and encoding adequate, informative and discriminative features to build effective machine learning models. In the field of malware detection, mainly two approaches coexists to extract features from an Android application: (i) static analysis and (ii) dynamic analysis. In the two next paragraphs, we review state-of-the-art studies based on these extraction approaches.

Static analysis feature extraction. In 2014, authors of Drebin [21] used a broad static analysis to collect a large set of features, both at the bytecode and the resource level of an application, such as permissions, suspicious api calls, application package name or component class names. The maliciousness of the application is then expressed by the presence or the absence of a given feature in the feature vector. Afterwards, authors use a linear Support Vector Machine algorithm to train a classifier. Authors demonstrated that Drebin can achieve an accuracy of 94% with their dataset composed of 123 453 benign applications and 5560 malware.

With DroidAPIMiner [20], Aafer et al. proposed a static analysis technique to detect the presence or absence of 8375 framework API calls in analysed applications. They train three different classifiers using DT, KNN and SVM algorithms and demonstrate that their approach is able to reach an accuracy of 99% on their dataset composed of 16 000 benign applications from the Google Play Store and 3 987 malware from McAfee and the Malware Genome Project [7]. However, the approach proposed by DroidAPIMiner is sensitive to the change of API methods in the framework when new versions of the Android platform are released. To solve this issue, Mariconti et al. proposed MaMaDroid [19], an approach that uses static analysis to extracts the entire application call graph and encodes it in an abstract representation by keeping only Java method package names. MaMaDroid produces feature vectors by calculating the probability for a method to be called by another in the call graph using Markov chains. To perform malware detection, authors train three classifiers with a RF, KNN and SVM algorithms and show that all

classifiers outperform DroidAPIMiner on 6 datasets built for the experiment.

Despite the good results obtained by the aforementioned studies, attackers started using new evasion techniques to defeat static analysis, such as *reflection* or *dynamic code loading*. Such techniques allow the attacker to dissimulate its malicious payload in places unreachable by a static analysis program or to make it unreadable. In this thesis we will focus in more detail on the effects of these new attacks on machine learning detection systems based on static analysis.

Dynamic analysis feature extraction. Other studies [32, 81, 82] proposed ML-based malware detection approaches using features collected with dynamic analysis. In 2012, Shabtai et al. [32] proposed Andromaly, a ML-based approach that use a dynamic analysis to sample various device metrics such as CPU load, memory usage or power usage to detect suspicious or abnormal activities. Tested with decision trees (DT), K-nearest neighbors (KNN) and Support Vector Machine (SVM), the approach allowed to successfully detect 4 malware among 44 applications tested. With Madam [83], authors proposed an on-device malware detection framework implemented at the framework level. The approach dynamically monitors system calls, frameworks API calls and network traffic of running applications and encode generated traces into a feature vector. Afterwards, they train a KNN classifier with generated feature vectors and use it to detect malicious behaviours occurring on the device. Authors of Madam claim to effectively block 96% of malicious applications on their dataset of 2800 malware. In 2017, Lindorfer et al. proposed Marvin [82], a ML-based approach that combines static and dynamic analysis to extract a wide range of features, such as permissions, phone activity, network traffic, crypto operations, etc. Authors use a SVM algorithm to train a classifier and evaluate it on a dataset of 135 000 applications including 15000 malware. The trained classifier reached an accuracy of 98.24%.

While more and more malware detection related works used dynamic analysis to extract features in recent years, they remain a minority because of the inherent scalability issues of the approach. The dynamic analysis of an application requires to execute it on an Android device to observe and monitor its behavior. Therefore, to make such analysis scale for thousands of applications, applications are often executed in virtual environment. Unfortunately, recent malware can detect at runtime that they are running in a virtual environment [28]. For these reasons, static analysis remains the approach mainly used to extract the features of an application.

Limits of machine learning-based detection

Contrary to signature-based approaches, ML-based approaches are more likely to detect malware variants generated by attackers. This is because ML-based approaches train classifiers able to generalize the malware detection problem from the data used to train it instead of comparing signatures together. However, recent studies [39, 84] have raised a number of concerns on the generalizability of reported results in the literature. Performance of machine learning based detection systems are highly dependent to the data used to train them. Unfortunately, publicly available malware datasets are extremely rare [7, 21, 40] and are criticised to be unreliable and unrealistic [23, 35]. In particular, existing datasets are composed of nonfunctional, outdated samples [7] that do no fit with the current malware evolution. Moreover, ML-based detection approaches [21, 37–39] often rely on antivirus aggregators such as VirusTotal [42] to build large malware datasets. Antivirus aggregators are online platforms that run many antivirus products from third-party security companies to vet files uploaded by users. This approach may not be accurate because (i) samples flagged malicious by aggregators are not always truly malicious and (ii) malicious samples uploaded for the first time on VirusTotal are often wrongly flagged as benign [23].

To solve the problem, several studies [23, 85, 86] proposed new approaches to build malware datasets without relying on a third-party oracle. In 2015, Maiorca et al. [86] used ProGuard [87] to build malware variants by obfuscating malware samples from the manually curated *Contagio* dataset [88]. With RmvDroid [23], authors create new malware datasets by selecting applications that are both flagged malicious by VirusTotal **and** removed by Google from the Google Play Store. Unfortunately, the datasets generated by these studies remain little used because they do not offer sufficient diversity in the samples they contain to allow a model to generalize malware detection compared to existing datasets. Moreover, even if samples collected by RmvDroid were removed from the Google Play Store and vetted by VirusTotal [42], there is still no evidence that these applications are real malware. Google may have removed theses applications for other reasons such as non-compliance with Play Store's terms of use. In this thesis, we introduce new concepts that will help to increase the quality of application datasets used to train and test machine learning-based detection systems.

Another problem of existing malware datasets used to train ML-based scanners is that they lack corner case malware samples that use complex obfuscation techniques to hide

their malicious payload. For example, malware samples that leverage *packing*, *reflection* or *dynamic code loading* are problematic for machine learning based approaches because they prevent them to use feature that truly express the application's behaviour. The problem is even more accentuated by the fact that trained model are *drifting* from reality, as recent studies show [35, 39]. The concept of *model drift* highlights the fact that datasets used to train learning models previously cited studies are not close enough to reality. Firstly, used datasets suffer from *spatial bias*, i.e. the proportion malware in datasets is not equal to the proportion of malware in the wild. With TESSERACT [39], authors suggest that 10% of applications in the wild are malware, whereas most of studies use balanced datasets with 50% of malware. Secondly, used datasets suffer from temporal bias, i.e. in proportion, the age of samples within datasets do not represent correctly the age of samples found on stores.

Such limitations have opened a way for a new range of studies towards the generation of Adversarial Examples for classifiers. The main aims of such studies is (i) to challenge existing ML based detection systems with corner cas malware samples and (ii) to synthetically create realistic malware variants to increase the size of malware datasets.

2.3 Adversarial Attacks to Malware Detection

Scanners built to detect malicious behaviours are used in various domains such as Desktop antivirus, spam filters, DoS attacks detection, malicious pdfs or malicious applications on mobile detection. Over the years, various studies have shed the light on the vulnerabilities of such detectors and lots of efforts have been made to identify and taxonomise related attacks [7, 89]. Among them, two major attacks, *poisoning* and *evasion* attacks have been practically demonstrated. Poisoning attacks consists of introducing inaccurate and/or evasive data into the data used as ground truth for a scanner, for example to train a machine learning-based classifier. Such attacks have been successful in several domains such as malicious pdf detection, spam filters and handwritten digits detection [90–92]. Evasion attacks which cause misclassification of new data in the testing phase, have also been successful against pdf detectors [93] or biometric authentication systems [94]. Limitations of those attacks is that it requires a deep knowledge of the adversary as well as access to the training set used to train the classifier, or the ground truth data for signature-based detection. To overcome these limitations, new studies in the field have proposed more realistic evasion attacks that assume less knowledge about

the targeted detection system [95, 96].

2.3.1 Review of existing studies

Evasion attacks on malware scanners can be thought as the action of crafting Adversarial Examples (AE), that said corner case samples specifically created to evade a targeted classifier. Generation of adversarial examples is discussed in a large body of literature [33, 97, 98]. Crafting AE has been mostly successful in continuous domains, such as images [99], where with small perturbations it is possible to change its classification while being invisible from the human eyes. However, in constrained domains, where the classification relies on machine learning and domain specific feature vectors, perturbations on feature vectors may lead to corrupted samples. As such, several studies [100, 101] perform attacks on multiple machine learning and deep learning based scanners by applying perturbations at the feature vector level. While these attacks are theoretically successful, authors are not able to track those perturbations back to create fully working adversarial examples. While some success have been recorded for PDF classification [34, 102] few attempts have been made to generate functional Adversarial application samples for Android malware classification.

The tension between realistic and effective adversarial attacks.

Regarding the Android ecosystem, multiple research works have studied the security problems of malware detection through adversarial attacks [33, 103–107]. Several of these approaches are mainly producing AEs using obfuscations techniques on subsidiary files such as Manifest files (e.g. permissions), or at the bytecode level with method renaming, method call reordering, package renaming or string encryption. Such obfuscation techniques are sufficient to evade signature-based scanners but fail against more advanced ones as the behaviour of crafted AEs remain accessible to static analysis tools and therefore still interpretable. Nevertheless, in pure academic exercices, researchers have shown how manipulating feature vectors can lead to samples that can evade detection even by sophisticated detectors [100, 101, 103, 108]. In general, however, it can be infeasible to track those changes back to a sensical object. While perturbations at the feature vector level are theoretically effective, they do not add much value to finding proactive solutions to the problem of ever-evolving Android malware variants [22, 109].

Results of the aforementioned studies highlight the tension between the *realism* and

the *effectiveness* of adversarial attacks. A *realistic* attack can be thought as an attack that tries to imitate a real world attacker with respect of the following conditions:

— **Working adversarial examples.** An attacker will try to craft working malware variants that evade detection scanners and trigger their malicious behaviour when executed on a target device. Hence, attacks that do not allow to create functional adversarial examples, such as perturbation at the feature vector level, cannot be considered realistic.

— **Attack against a black-box system.** An attacker is unlikely to have a deep knowledge about the targeted detection scanner, such as the algorithm used to train a ML-based model, or the data used to train it. As such, to be considered realistic, adversarial attacks must treat the targeted detection systems as a black-box.

Besides, an attack can be considered *effective* when samples generated are misclassified by a targeted detection scanner. In this thesis, we are particularly interested in finding the right compromise that allows us to highlight the vulnerabilities of Android malware detection scanners while making attacks practically feasible.

Study	Year	Vector only	Number of generated samples	Number of working samples	Black-box attack	Evaluated approaches
Android HIV [33]	2015	No	4 560	Unknown	No	Drebin (SVM) [21], MaMaDroid (RF,SVM,KNN) [19]
Mystique [110]	2016	No	10 000	30	Yes	VirusTotal [42], 9 academic solutions [21, 51–53, 111–115]
Mystique-S [104]	2017	No	44	44	Yes	12 mobile antivirus, TaintDroid [52]
Yang et al. [106]	2017	No	2 612	Unknown	Yes	7 engines from VirusTotal, AppContext [116],Drebin [21]
Demontis et al. [22]	2017	No	10 500	Unknown	Yes	Drebin [21], custom improved SVM
Pierazzi et al. [109]	2019	No	5 330	Unknown	Yes	Drebin [21], custom improved SVM proposed in [22]
Grosse et al. [103]	2017	Yes	-	-	No	Custom DNN
Grosse et al. [100]	2016	Yes	-	-	No	Custom DNN
MalGan [101]	2017	Yes	-	-	Yes	Custom DNN
Dillon [108]	2019	Yes	-	-	Yes	Custom DNN

Table 2.6 – Overview of studies regarding adversarial attacks on Android antivirus

In this logic, several works (see table 2.6) studied more complex approaches to find good a trade off between realistic and effective adversarial attacks against Android malware scanners. These studies can be compared according to several criteria that make it possible to quantify the *realism* and the *effectiveness* of the attack: (i) the number of adversarial examples generated, (ii) the number of working samples among them, and (iii) the assumed knowledge about target systems. Over the ten studies listed, six of them proposed an approach that generates real adversarial examples in the form of Android APKs.

With Android HIV, authors perform two distinct attacks on the MaMaDroid [19] and Drebin [21] ML-based detection methods. To perform the attack on MaMaDroid, they rename classes and add an arbitrary number of method calls in the application bytecode. To perform the attack on Drebin, they add permissions to the *AndroidManifest.xml* files and API calls in the application bytecode. In the experiments, authors successfully generate 4 560 malware variants from existing malware from which 86% and 99.4% evaded the MaMaDroid and Drebin approach respectively.

While authors claim to perform a black-box attack on the targeted scanners, they perform two different attacks specifically tailored for each scanner, suggesting that attacks have been designed with knowledge of detection systems internals. In addition, authors do not provide evidence that the attack is still effective when targeted ML scanners are re-trained with adversarial examples generated by their approach. Finally, authors do not mention if any experiment has been done to determine if generated malware variants are functional, which make attacks potentially *unrealistic*. In this thesis, we present new approaches that does not target one particular scanner but all at the same time. We further address the problem of adversarial re-training for machine learning-based scanners. We finally present new alternatives to automatically ensure that synthetically crafted variants are fully working.

With Mystique, Meng et al. [110] propose an approach to evaluate the robustness of VirusTotal, and 9 academic solutions including the Drebin [21] machine learning scanner by generating malware variants from 4 attacks and 2 evasion techniques found in existing malware samples. To do so, they generate 10 000 malware variants for each attack and evasion feature from 1 260 samples randomly selected from the MalwareGenome dataset [7]. Authors show that several malware variants can bypass academic solutions and antivirus engines on VirusTotal achieve a detection ratio of 30% on average against uploaded malware variants. In the paper, authors manually tested 30 malware variants over the 10 000 generated. Mystique-S [104] is an evolution of Mystique where authors add a *dynamic code loading* evasion feature to 44 generated malware variants. Authors show that all variants created are working by manually vetting them and succeeded to bypass 9 mobile antivirus products.

Contrary to Android HIV, these two studies show that some of generated malware variants are fully functional. This verification is however made only on a small sample of all generated variants (30 over 10 000). Moreover, the 30 samples tested are generated using a simplified approach of the attack presented by authors, which does not guarantee that

evading malware variants crafted with the full approach are actually working. Especially, original malware samples used to create variants come from the MalwareGenome [7] dataset, which mainly contains outdated and un-working samples. Another drawback of the study, similar to Android HIV, is that authors do not give evidence that their attack is still effective after re-training machine learning-based scanners with malware variants they generated. In our work, to guarantee *realistic* adversarial attacks, we ensure the original malware chosen to craft adversarial examples are up to date and fully working.

In 2017, Yang et al. [106] proposed a novel method to generate malware variants by using the concept of code transplantation. They perform a semantic analysis on existing applications and generate variants by migrating (i.e. transplanting) portions of code from one place to another without changing the behaviour of the application. With this approach they succeed to produce 2 612 malware variants. Over all generated variants, 178 successfully evaded the AppContext [116] scanner, 38 evaded the Drebin [21] scanner, and 565 evaded 7 antivirus engines chosen on VirusTotal.

While authors succeeded to generate adversarial examples with their approach, most of generated variants are still detected by one or more detection scanners. Moreover, authors do not mention if successful adversarial examples are evading only one target scanners or all of them. Similarly to other studies, authors do not say in their evaluation process of generated malware variants are functional.

More recently, Demontis et al. [22] proposed a new attack model to defeat the malware detection method proposed by Arp et al [21] with Drebin. In their approach, they describe seven different attacks strategies applied on a dataset of 1 500 malware from the *Contagio* dataset [88], totalling 10 500 variants:

— *Zero effort attack.* No manipulation performed (ground truth)
— *DexGuard obfuscation attacks.* This scenario includes three attacks. Malware are obfuscated by the `dexguard` obfuscator tool, performing string encryption, reflection and class encryption.
— *Mimicry attack.* This scenario represents an attack where the attacker know in advance the training dataset and therefore the feature space.
— *Limited knownledge attack.* This scenario represents an attack where the attacker know in advance the training dataset and the learning algorithm used by the targeted system.
— *Perfect knownledge attack.* This scenario represents an attack where the attacker

has access to the trained classifier of the targeted system.
Authors performs these seven attacks on two classifiers, a linear SVM similar to the one used in the DREBIN [21] study and a custom *secured* SVM, supposed to be more robust to such adversarial attacks. The experimental results show that the *Zero effort attack*, *DexGuard obfuscation attack* did not affect classifiers performances. Authors further show that the original DREBIN classifier is vulnerable to *mimicry*, *limited knowledge* and *perfect knowledge* attacks and that the custom *secured* SVM is ten times better.

Following the study of Demontis et al. [22], Pierazzi et al. [109] proposed a new attack, constrained to fit with the problem-space, that is applying transformations to a malware while ensuring that the resulting adversarial variant is functional. To do so, they formalize an approach that describes the possible perturbations applicable at the resource and bytecode level without breaking the application. They evaluate their attack on the DREBIN [21] and the custom classifier proposed by Demontis et al. [22]. Authors demonstrate a misclassification rate of 100% on both trained classifiers for the 5 330 malware variants generated with their approach.

However, authors are only targeting two detection approaches (DREBIN [21] and the custom *secured* SVM) and omit to test their approach on detection systems such as MAMADROID [19], APPCONTEXT [116] or commercial antivirus products. Typically, the detection method proposed in MAMADROID [19] is different from DREBIN [21]'s as it encodes call graph features using a markov chain algorithm to infer the application's behaviour, instead of raw binary encoded contextual features. It is therefore hard to know if the attack presented would be effective against other scanners built with different detection methods. In our work, we claim to be more exhaustive in our experiments by evaluating different scanners based on different detection methods to show their limits against our adversarial attack method. Besides, authors test if generated malware variants are functional after applying perturbations, authors automatically install and run them on Android emulators.

2.3.2 Open problems

The review of aforementioned studies show that either one or several key problems are not addressed in the field of adversarial attacks against Android malware scanners.

Attacks against black-box scanners. In several cases [22, 104, 109, 110], adversarial attacks proposed are performed with partial or full knowledge of the targeted system,

i.e. with knowledge about the learning algorithm used and/or the datasets used to train the model. However, it remains unclear [22] how an attacker can evade existing machine learning detection systems with absolutely no knowledge about the target system. To address this problem in our work, we investigate the effectiveness of a new adversarial attacks by crafting adversarial examples in the dark, i.e. by considering each evaluated scanners as a black-box.

Attacks are effective but not realistic. To the best of our knowledge, no study proposed an adversarial attack scenario for which adversarial examples generated have proven to be *realistic*, that said where variants generated conserve the malicious behaviour of the original malware despite the perturbations operated on it. Most of studies [100, 101, 103, 108] perform only perturbations at the feature vector level, without being able to reflect them on an application file. Some studies [33, 109] propose an automated mechanism to install and run malware variants but do not give evidence that the variant effectively triggers its malicious behaviour and does not crash. Such results suggest that more efforts are required to find evidence that such adversarial attack are effectively viable in practice. The review of aforementioned studies highlights a trade-off to consider between the probability of hitting an adversarial example, i.e. a malware variant that successfully bypass a detection system, and the likelihood that generated sample is realistic (i.e functional). In this thesis, to address the lack of *realistic* adversarial examples we propose KillerScreenshot (see chapter 5), a tool that enables to ensure that a crafted malware variant is triggering its malicious behaviour at runtime by comparing its screen activity with the original malware.

Few works studied attacks on commercial antivirus. Adversarial attacks against commercial antivirus remain sought [37, 106, 110] and often target few antivirus engines empirically chosen. So far, related studies seems unsuccessful in reaching 100% misclassification against antivirus engines hosted on VirusTotal. Moreover, to the best of our knowledge, only one study [104] tried to evade fully-fledged mobile antivirus products, but only with 44 malware variants. We believe that more efforts are needed to find example of attacks, especially with more complex evasion techniques, that challenge commercial antivirus in order to strengthen them against future real world attacks. To address this problem in our work, we thoroughly evaluate our adversarial attack approach against all commercial antivirus engines running on VirusTotal. Moreover, we propose a new

tool, namely KillerTotal, to evaluate the robustness of fully-fledged antivirus mobile applications.

Evasion methods remain simple and predictable. In reviewed studies, methods to create adversarial variants are often using obfuscation, perturbation at the vector level or perturbation at the bytecode level but do not investigated more complex attacks leveraging on more recent and complex evasion techniques such as *dynamic code loading* or *bytecode encryption*. Only one study [104] actually studied an attack using *dynamic code loading* by creating 44 malware variants using this technique. These evasion attacks are however heavily used by attackers to craft malware [117]. To properly audit malware detection scanners, we believe that adversarial attacks must integrate such more complex evasion techniques to cover as much corner cases as possible to create effective challenging datasets. To address this problem in our work, we propose KillerDroid, a toolchain that combines complex evasion techniques to create adversarial examples.

Despite the remaining open problems, Adversarial attacks against Android malware detection systems have however proven to be successful under certain circumstances. To fight against this emergent threat, new works started investigating new methods to reinforce detection systems against adversarial attacks.

2.4 Defenses against adversarial attacks

To reinforce malware detection scanners, especially machine learning-based, several research works [22, 100, 108, 118] (see table 2.7) investigated *adversary-aware* approaches, that said designed to be more resistant against evasion.

Study	**Year**	**Attack use real samples**	**Re-Training**	**Secured approach**
Dillon [108]	2020	No	No	Custom DNN
Podschwadt et al. [118]	2019	No	Yes	Custom DNN
Grosse et al. [100]	2016	No	Yes	Custom DNN
Demontis et al. [22]	2017	Yes	No	Drebin (SVM)

Table 2.7 – Overview of studies regarding defenses against adversarial attacks

Securing deep neural networks

Recently, several studies proposed approaches to secure detection systems built upon deep neural networks. In 2016, Grosse et al. [100] investigate three techiques to improve the detection rate of a neural network-based scanner: *feature reduction*, *Distillation* and *re-training*. *Feature reduction* consists in considering less features to train the model in order to decrease the model's sensitivity to changes in the input. *Distillation* allows to transfer knowledge from a large neural network to small ones in order to increase the generalization performances of smaller networks. Finally *re-training* consists in re-training the neural network model with a training dataset that includes adversarial examples. In their evaluation, Grosse et al. conclude that *feature reduction* and *distillation* do not improve much the malware detection rate of the tested neural network but *re-training* had a significant impact, lowering the missclassification rate from 82% to 67% in best conditions. In 2019, Podschwadt et al. [118] proposed four defenses methods against twelve different attacks performed on detection systems based on deep neural networks. The defenses evaluated include *distillation*, *adversarial training*, *ensemble adversarial training* and *random feature nullification*. In their evaluation, authors found out that the *distillation*, *random feature nullification*, *ensemble adversarial training* and have almost no positive impact on adversarial robustness. However, similarly to Grosse et al. [100], authors concluded that the *adversarial training* defense, which consists of retraining the classifier with adversarial examples, provided a significant robustness. Finally, Dillon [108] proposed another *adversarial training* defense by training a deep neural network with adversarial samples specifically generated to program the network in order ignore the number of benign features and focus on the presence of absence of malicious features. To generate malware variants, authors randomly add benign features to a malware feature vector. They first demonstrate that their attack succeeds on a trained deep neural network and then demonstrate that they can increase the robustness of the network by training it with obfuscated samples.

The aforementioned studies suggest that re-training a deep neural networks with labelled adversarial examples is an efficient way to strengthen a detection approach targeted by an adversarial attack. Other studies in the domain of PDFs malware suggest that other machine learning methods may also benefit from adversarial re-training to increase the robustness of their models. However, as reviewed in the previous section, no study investigating adversarial attacks on Android malware scanner have evaluated their approach in case of adversarial re-training of the target systems. In this thesis, we present a new

adversarial attack together with a robust evaluation approach that takes into account the concept of adversarial re-training for all machine learning-based scanners evaluated.

Securing existing ML-based scanners

Demontis et al. [22] proposed a novel method to increase the robustness of the DREBIN [21] detection system based on a Support Vector Machine (SVM) learning algorithm. To do so, they formalize an approach that improve the security of a linear classification by enforcing a learning on more *evenly-distributed* features weights. This requires an attacker to manipulate more features to evade the classifier detection. In their evaluation authors show that their improved classifier is indeed *ten times* more efficient than the original DREBIN approach to deal with adversarial attacks they performed. However, their improved approach have been defeated by a new study [109] (see section 2.3.1).

2.5 Conclusion

In this chapter we reviewed many works that propose new approaches to make the Android platform more secure. These studies show the effectiveness of a number of mechanisms to prevent a malicious application from accessing user's sensitive data without its consent. Other studies show that a stronger application isolation can prevent a malicious application from gaining excessive privileges on the system. However, this enhanced security must not come at the expense of the proper functioning of legitimate applications. Therefore, great control is left to the user: he must choose, particularly through the permissions system, to grant or deny certain privileges to an application. This can lead the user to misinterpretation on application's intentions, and he may accidentally allow a malware to perform its malicious behavior. This is all the more difficult as malicious and legitimate applications can look similar in many ways.

To fight this phenomenon, many studies have emerged to try to distinguish a legitimate application from a malicious one. These studies use static and dynamic analysis to collect information on tested applications and then mechanisms such as signature verification or machine learning to classify tested applications as benign or malicious. Over time, these mechanisms become more and more efficient, with an accuracy that can exceed 99However, the evaluation of these approaches is made under specific conditions, with predefined datasets, sometimes obsolete which do not reflect reality.

In order to assess the effectiveness and challenge these detection scanners with more precision, new studies have been working on adversarial attacks. Adversarial attacks consist in generating malware variants capable of escaping detection systems. Several studies have shown that this practice is effective and have succeeded in challenging existing detection systems. However, several problems inherent in constructing adversarial examples threatens the validity of the proposed studies. In fact, there is a tension between the *realism* and the *effectiveness* of an attack. Most studies perform an attack with knowledge of how the targeted system works, and the variants generated are not proven to be functional, which makes these attack *unrealistic*. In addition, the attacks carried out remain simple and predictable and do not take into account the new evasion techniques used by attackers to build malware today. In the following of this thesis, we will provide new approaches that aim at addressing the aforementioned open problems, especially by improving both the realism and the effectiveness of adverarial attacks against Android detection systems.

Chapter 3

Exploring malware detection tools on Android

In this chapter, we present our works dedicated to improve the performances of existing machine learning based detection approaches in the Android ecosystem. We begin the chapter with background knowledge regarding the Android platform and especially Android applications. Then, we review the techniques used to perform program analysis on Android applications. We further discuss the limits of existing techniques to perform malware detection. Finally, we present DroidAutoML and Groom, two frameworks designed to increase the accuracy of machine learning based detection approaches on Android.

3.1 Background on the Android ecosystem

In this section, we provide background and definitions related to the Android platform to contextualize our work. Specifically, we discuss in more details several security limits of the Android platform.

3.1.1 Android platform architecture

The Android platform is an open-source project built on a linux kernel intended to run on multiple types of smart devices [119]. The software stack can be roughly divided in 5 distinct layers. The lowest layer is the linux kernel, responsible for managing hardware resources and allowing the development hardware drivers for different types of devices. Above the linux kernel is the *Hardware Abstraction layer*, that exposes device hardware capabilities to the Java framework API (see below). The third layer is the Android runtime (ART). Each application running on a device is running in its own instance of ART which allow to execute dex binary files, a bytecode format specifically designed for Android.

Java source code can be compiled into dex bytecode. The fourth layer is the Java API framework. All features of the Android operating system are available through a unified API written in Java. The Java API framework is the main interface used by application developers to use features and interact with the system. The API framework proposes groups of Java methods calls to handle services such as: the View and display system, access to external resources files, manage application activities or access to data from other applications. The fifth layer represents system and third-party applications running on the device.

3.1.2 Android application architecture

Android applications consist of specific Java programs executed by the Android operating system. A normal desktop Java program is executed by the runtime environment as a standalone application and remains independent throughout its execution. On the contrary, the Android runtime subordinates the apps it executes, which allow the Android operating system to pause and resume the execution of an app at any time. Such mechanisms are implemented to comply with the restrictions of mobile devices. Smartphones are composed of limited shared resources such as storage, memory or power that must be shared with multiple running processes. As such, the operating system prioritizes access to resources to the foreground application rather than unused applications in the background.

Application sandbox. The Android platform leverages on Linux user-based protection to isolate application resources. In the Linux system, each user is assigned a unique user id (UID), and users are isolated by the kernel in such a way that one user cannot access the data of another user. On Android, the system assigns a unique user id to each Android application and runs it in its own process. This mechanism allows to isolate applications from each other and protects the system from potential malicious behaviours. The Android sandbox is in the kernel, thus if one malicious application tries to read the data of another application, this is not permitted as the malicious application does not have user privileges.

Over years, several works [7, 67] demonstrated that the application sandbox model is not invulnerable, which drove the support of new security features such as SELinux [13, 60]. Since Android 5.0, SELinux enforces mandatory access control (MAC) over all process, which allow to better control access to application data and better protect users from potential vulnerabilities in application's code.

Components. To develop an application, developers implement Java classes that inherit from predefined system classes called *Components*. Each component has callbacks that can be overwritten by the developer to allow the application to react to events emitted by the system. The Android documentation defines four types of components that drive the application *lifecycle*. The *Activity* component can be thought as a task coupled to the graphical user interface. The *Service* component are used to perform long-running background tasks that must not interfere with user interactions to preserve user experience. The *Broadcast Receiver* component is responsible for listening and react to events sent by the system or other running applications. Finally, the *Content provider* component provides an interface to the application database to store and access application specific data. Application components implemented by the developer must be declared in the `AndroidManifest.xml` file so that the system is aware of them and can interact with them. The manifest file acts as a declarative file where the developer must declare the intentions and capacities of its application.

Android application developers take advantage of these components to use hardware device capabilities, read and write user data and react to System wide events. These features give a lot a freedom to the executed application that can affect user privacy and security. For these reasons, Android has implemented several security locks to have greater control on app installed on device.

Permissions. To protect the privacy of an Android user, the operating system enforces an explicit permission model. An application cannot perform sensitive operation such as reading user contacts or sending SMSs without having the specific permission granted by the system. The application must declare all permissions it will ever use in the *AndroidManifest.xml* file. Until Android version 5 (Lollipop), permissions requested the application were presented to the user at installation time. If the user did not wish to grant permissions to the installed application, he could choose to uninstall it but he did not have the possibility to grant only part of the requested permissions. Due to this limited permission model, recent Android versions (from version 6) switched to a more granular model where the user can grant permissions separately [15]. The new permission model is runtime based, i.e. whenever the application needs to perform an operation protected by a permission, a modal is presented to the user to ask him to either allow or deny the application action.

Recent studies [49, 120] stated that this permission model is insufficient to protect

user privacy and prevent data leakage for several reasons. Firstly, applications are often *over-priviledged*, i.e. they request more permissions than required to perform their tasks. Secondly, users do not always carefully review permission requested by applications for fear of not being able to use the application correctly. Thirdly, permissions are coarse-grained, thus for example, if an application requests the permission to read the device internal storage, this does not restrict the files the app can read. Finally, a recent study [49] discovered several side channels in the Android framework that allow applications to access sensitive data without having the permission to do so. While studies [121, 122] proposed new approaches to refine the permission model and make it more fine-grained, attackers still take advantage on the aforementioned limitations to successfully execute malicious behaviours on users devices.

3.1.3 Malware infection in the Android ecosystem

To successfully perform an attack on a device, the attacker must first design a malicious application capable of taking advantage on vulnerabilities of the device and/or hijack device's functionalities for malicious purpose. Then, the attacker must reach the user's device by successfully installing his malicious application on it.

How attackers take advantage of the Android platform. By targeting smart devices with malicious applications, an attacker mainly seeks two goals: extort money from the user and/or spy on him. Thus, malware can take many forms, depending on their behaviour, their goal and the device's capabilities or vulnerabilities they exploit. To better detect these threats, researchers have characterized and categorised malware into families [7] and they seek to know how they reach smart devices [123].

For example, *ransomwares* encrypt the user filesystem and block the user interface until the user accepts to pay and ransom to obtain the decryption key to recover its files. To block the user interface, the *ransomware* can take advantage on the *ViewOverlay* [124] feature of the Android platform, which allow an application to display an extra layer that sits on top of all other views (see 3.1). Another example is malicious applications that leverage capabilities *Service* component to send SMSs to premium phone numbers in the background. Other families of malware leverage vulnerabilities at the kernel level to deploy *rootkits* and perform privilege escalation to gain *root* access on the targeted device [8].

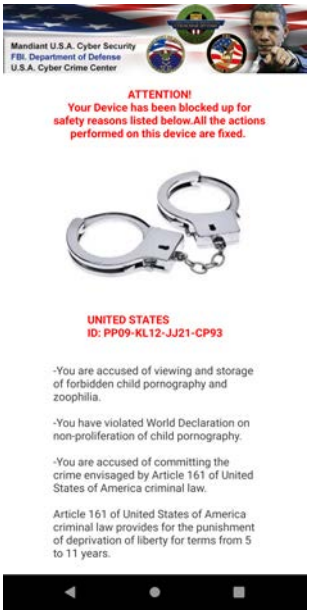

Figure 3.1 – Example of a Ransomware displaying an un-closable overlay layer on the user interface (Android 9.0)

How attackers evade detection systems. Mainly, attackers try to design the malware in such a way that it is difficult to differentiate it from a legitimate application, to prevent it from being detected before it is executed on the target device. The following list gives three examples of techniques used by attackers to evade detection systems (an in depth study of malware evasion techniques is provided in section 4.1.1):

— **Obfuscation.** Obfuscation consists in making the code of an application difficult for a human to read and understand, mainly to avoid reverse engineering.
— **Packing.** Packing is the action of hiding a malicious payload within a host application. Attackers may hide their malicious payload into an existing legitimate application to make the resulting application looks legitimate.
— **Dynamic code loading.** This technique consists in leveraging a functionality of the Android runtime to load bytecode payloads at runtime from various locations, such as device local storage or over network.

To thwart such evasion techniques used by attackers, Android security actors have developed automated software analysis approaches.

3.2 Program analysis

Program analysis, which consists in automatically analyzing the behaviour of a program, is used in a wide range of applications such as correctness [125, 126], robustness [127] or safety [128, 129]. Since the emergence of malware on mobile devices, program analysis has become the main method for detecting malicious applications [21, 32, 82, 130]. Regarding malware detection on mobile devices, application analysis can be roughly categorized into techniques using static and/or dynamic analysis.

A static analysis allows to quickly analyze binaries of an application without having to execute it, in particular without having an negative impact on devices resources. However, static analysis is limited to the analysis of the visible part of the application's bytecode. Consequently, attackers may use advanced obfuscations techniques such as *dynamic code loading* or *encryption* to put their malicious payload out of range of static analysis. Due to these limitations, dynamic analysis is often explored as an alternative for analyzing applications in real time [81, 131]. Dynamic analysis is performed by executing the program on a real or a virtual machine. If performed with a sufficient number of test inputs, dynamic analysis may help to uncover runtime values of an application and defeat advanced obfuscation techniques. To overcome scalability issues [132], dynamic analyses are often performed in parallel in *virtual environments* [133], which make them vulnerable to evasion attacks [28].

3.2.1 Limits of dynamic analysis to malware detection

Several works [81, 132] demonstrated that dynamic analysis can be very efficient towards detecting malicious behaviours that can not be detected by a static analysis. On Android, most known works [52, 134] monitor the application runtime in a custom virtual environment to perform dynamic taint analysis. DroidScope [134] performs the dynamic taint analysis at machine code level, whereas TaintDroid [52] monitors installed applications to detect if private-sensitive data is leaving the device. While these studies are not directly oriented toward detecting malware, they are helpful to shade the light on potential unwanted behaviours that occur at runtime. More recent studies such as MADAM [81] and Marvin [82] proposed approaches to detect Android malware using heuristics generated using dynamic analysis such as *network operations*, *file operation* or *phone events*.

Although techniques using dynamic analysis are promising, several limiting factors prevent them from being deployed at an industrial scale to fight against the high volume

of unforeseen malware in the wild. Typically, to make it scale and for isolation purposes, such analyses are executed in virtual environments. Unfortunately, recent malware can detect at runtime that they are running in a virtual environment [28]. Moreover, malware are known to implement evasion techniques such as *logic bombs* and *time bombs* which allow them to stay inconspicuous during analysis. Logic bombs can be thought as *malicious application logic* that is executed only under certain circumstances. For example, some malware implement malicious payloads that will only be executed when the device is located at a given geographic position. Similarly, *time bombs* are malicious application logic that enable the malware to trigger its malicious actions only after a certain amount of time or periodically, at specific hours.

Accordingly, dynamic analysis suffers from scalability issues, and is rarely used due to its strong requirements both in terms of time and resources. To overcome these limits and perform malware detection of a large volume of application, static analysis is often chosen over dynamic analysis.

3.2.2 Advantages of static analysis in malware detection

For the reasons listed above, majority of studies in the field of malware detection prefer approaches leveraging on static analysis. Static analysis does not require to execute the application on a device, thus it is less expensive to analyse large volume of applications. Another important aspect is that static analysis not only to analyse the bytecode of the application, but also the resources associated with it. Thus, several studies [21, 135] correlate heuristics from both resources and bytecode to identify potential threats and improve their detection rate. Finally, static analysis makes it possible to abstract the execution environment, the interactions with the user as well as the semantics of the program, which makes it possible to identify problems according to various parameters. As such, where dynamic analysis can only analyse one execution at a time, static analysis can perform an exhaustive search on all possible application execution paths.

Although static analysis is the most popular approach regarding malware detection, it is not without flaws: various obfuscation techniques [105, 136, 137] flourished in the recent years to escape from it such as *class and method renaming*, *reflection* or *dynamic code loading*. Code obfuscation is the process of altering an executable so that it is hard to reverse engineer while remaining fully functional. Code obfuscation is used by both developers of legitimate applications to protect their intellectual property and by attackers to dissimulate the malicious intentions of their malware. Such practices makes it harder

to distinguish legitimate applications from malicious ones. Therefore, for static analysis to remain effective regarding malware detection, more and more information is extracted from applications [21] and researchers rely on more robust techniques such as machine learning [19, 20, 138, 139].

3.3 Techniques to perform malware detection at scale

To respond to the dramatic proliferation of malware, the academic domain and the industry have witnessed a tremendous activity around designing antivirus scanners. Scanners analyse programs or applications and compares patterns found in code with information stored in database or submit them to trained machine learning models. After analysis, scanners gives a binary response by flagging the application either as *benign* or *malicious*, sometimes accompanied by a confidence score. This binary output can be used by market places or on-device security protections to decide whether to discard or keep the application.

Traditional detection products based on application signature are omnipresent on the security market [42, 140]. However, due to the rise of malware, these scanners become ineffective against emerging threats. Attackers leverage very complex techniques such as *polymorphism* [27] to create new malware samples and have their own tooling to verify if newly created samples are detected. In fact, malware authors can generate new malware variants with unforeseen signature at a high paced rate to bypass such signature based antivirus.

This limitation strengthens the need for more intelligent systems to proactively detect unseen 0-day malware. Machine learning models have then been extensively investigated to allow the identification of malware features and the scaling of malware variants identification [19, 21, 141].

3.3.1 Signature based detection

Due to the increasing number of applications published on public application markets, the need for quickly analyze incoming new applications has been on the rise. To do so, most of traditional antivirus detect malware by relying on technique known as signature-based detection. With signature-based detection, the decision whether to label an application as legitimate or malicious is done by comparing information such as a

computed file hash or bytecode patterns with sets of heuristics and rules generated from previously seen malware. As soon as a new malware is discovered in the wild, antivirus software companies put their hands on, compute their related file signature and add it to their ground truth database. Later on, antivirus compute signatures of files to be analyzed and compare them with previously stored signatures to perform malware detection. Consequently, the detection quality (i.e. the accuracy) of such antivirus heavily depends on the capacity of the software company to keep its ground truth database up to date as much as possible. If done correctly, the signature based technique remains an efficient and fast way to catch already known malware. However, keeping the ground truth database up to date requires a lot of manpower to analyse unknown samples as well as maintaining centralized platforms such as VirusTotal [42] to continuously collect a large volume of files. However, with the rise of cybercrime, these methods become unrealistic. Using *polymorphism* [27], attackers create new malware variants at a high paced rate with code transformations that confuse signature-based scanners. This inability to detect malware variants with unforeseen signatures pushed the need for novel systems capable of detecting unseen 0-day malware.

3.3.2 Machine learning assisted detection

Machine learning has been a breakthrough in the field of malware detection [142, 143]. Machine learning (ML) based scanners leverage a mathematical model built on sample data (known as training data) to perform prediction and malware detection. Most of ML scanners [19–21] are based on supervised learning: a set of labelled training samples are used to *teach* the model to learn a general rule to map the incoming data to a specific output. To create training data, human experts examine the code, the structure and the behaviour of applications to extract valuable information (i.e. features) that differentiate legitimate from malicious applications. In the literature, many kinds of features exists, ranging from basic ones such as suspicious API calls or *n-grams* in the code to more complex ones describing the entire application behaviour such as control flow graphs. Historically, the vast majority of Android malware detection systems [20, 21, 138] rely on static analysis to collect such features without the need to actually execute the suspicious software. Nowadays, scanners such as Google Play Protect (previously Bouncer) with heavy computational power can run Android applications in dynamic analysis environment to observe the application's general behaviour and collect features such as access control, network traffic or input-output control.

To evaluate the accuracy of a trained ML model, one can ask it to make prediction on data unseen by the model at training time, and calculate the number of correctly classified samples. Once trained and assessed, models can be deployed in production to make prediction on real unforeseen data. However, the accuracy of a ML model depends on many parameters and correctly evaluating its efficiency is not an easy task [39]. First, the performances of a ML scanner will depend on its capacity to generalize the malware detection problem from a fixed ground truth dataset made of benign and malicious applications. If the training dataset is not adequately chosen for the problem to solve (e.g. classifying malware from legitimate applications), the model will be unable to achieve good results on unknown data. Secondly, algorithms used to train the model can be tuned with a wide range of parameters (namely hyper-parameters) that can greatly influence its final performances. Finally, model performances are highly dependent to features chosen to describe and encode training samples. Several studies [22, 35, 39] have highlighted these limitation and paved the path for more accurate ML based malware detection systems.

3.4 Improving existing malware detection systems

To counter the rise of Android malware, security actors, from both research and industry, massively adopt machine learning techniques to improve detection accuracy. Although it is a first step towards improving detection, unfortunately, most of related studies neglect key aspects to build efficient detection models [35, 39]. In particular, it is commonly admitted in the machine learning domain, that performances of trained machine learning models depend strongly on several key aspects: (i) dataset quality, (ii) learning algorithms and hyper-parameter optimization and (iii) feature engineering. In this work, we propose two approaches to solve the aforementioned problems: (i) DroidAutoML, a framework to address the hyper-parameter optimization problem in ML malware detection and (ii) Groom, a framework extension of Soot that allow to improve the extraction of exploitable features for ML based detection methods.

3.4.1 DroidAutoML: hyper-parameter optimization for ML scanners

Hyper-parameter optimization is a well known problem in the machine learning domain [144–147]. Accordingly, the key underlying problem, usually referred as *Automated*

Machine Learning (AutoML) [148], is how to automate the process of finding the best suitable configuration to solve a given machine learning problem. As of today, no attempts have been done towards improving Android malware detection systems based on machine learning algorithms. Whether one [20, 21] or several algorithms [19, 32, 149] are evaluated, the evaluations are always carried out empirically, implying a manual process with few and/or default hyper-parameter combinations. Testing various algorithms along with a large set of hyper-parameters is a daunting task that costs a lot both in terms of time and resources [150].

To solve this problem, we present DroidAutoML, a new approach that automatically performs an extensive and exhaustive search by training various learning algorithms with thousands of hyper-parameter combinations to find the highest possible malware detection rate given two inputs: (i) a malware dataset (2) a feature extraction method. DroidAutoML is both generic and scalable. Its genericity comes from its ability to be agnostic to underlying machine learning algorithms used, and its scalability comes from its ability to scale infinitely horizontally by adding as much as machines as required to speed up the processing. To achieve this aim, and leveraging our expertise in the field of Android malware detection, we have defined and deployed a dedicated microservices architecture.

Our contributions are as follow:

— We propose the very first *AutoML* approach, named DroidAutoML, to improve the accuracy of technics based on machine learning algorithms to detect malware on Android. With DroidAutoML, there is no need anymore to manually perform empirical study to configure machine learning algorithms.

— We provide a dedicated microservices architecture specifically designed to fulfill the needs for genericity and flexibility as required by the Android malware detection domain.

— We thoroughly evaluate our approach, and applied it to the state of the art solutions such as Drebin [21] and MaMaDroid [19]. We demonstrated that DroidAutoML enables to improve significantly their performances: detection accuracy has been increased up to 11% for Drebin and 10% for MaMaDroid.

Hyper-parameter optimization

Usually, the number of *hyper-parameters* for a given algorithm is small ($\leq$5), but may take both continuous and discrete values leading to a very high number of different values

and so of combinations. For instance, common hyper-parameters include the *learning rate* for a neural network, the C and *sigma* for SVM, or the K parameter for KNN algorithms. The choice of hyper-parameters can have a strong impact on performances, learning time and resource consumption. As a result, *Automated hyper-parameter search* is a trending topic in the machine learning community [151, 152]. Currently, grid search and brute force approaches remain a widely used strategy for hyper-parameter optimization [153] but can require time and computational resources to test all possibilities. To deal with this issue, several frameworks are able to efficiently parallelize grid-searching tasks on a single machine, but this does not scale with the ever growing search space [154, 155].

3.4.2 DroidAutoML approach

DroidAutoML relies on a microservice architecture that separates concerns between data processing (feature *selection, extraction* and *encoding*) and training optimization ML models. Such a design enables DroidAutoMLto scale and stay agnostic to the evaluated scanner.

Microservices dedicated to features operations. *Feature extraction* and *encoding* are both operations specifics to each scanner. As such, each scanner has its own dedicated microservice for performing these operations (Figure 3.2, ■). We define k as the number of applications to process for a given dataset. For n different scanners, $n * k$ instances of $\blacksquare_{(i,j)}$ microservices with $i \in \{1..n\} \wedge j \in \{1..k\}$ will be deployed. Each $\blacksquare_{(i,j)}$ instance takes as input an *apk* to generate its corresponding features vector, interpretable by any machine learning algorithms. The generated *feature vector* is then stored into the *feature database* microservice (See figure 3.2,★).

Microservices dedicated to model training. *ML model training* operations are specific to a classification algorithm and the set of *hyper-parameter* used to parametrize it. Therefore, each algorithm has its own dedicated microservice to perform the training and testing of a model for one *hyper-parameter combination* (see Figure 3.2,▲). For l different algorithms, l different kinds of m instances of $\blacktriangle_{(i,j)}$ with $i \in \{1..l\} \wedge j \in \{1..m\}$ will be deployed where m is equals to the number of *hyper-parameter combinations* to test for a given algorithm. This allows to scale horizontally by spreading the workload across the available nodes in the cluster. A ▲ microservice takes two inputs: (i) a feature vector matrix from the feature database ★, and (ii) a set of hyper-parameter values. ▲ microser-

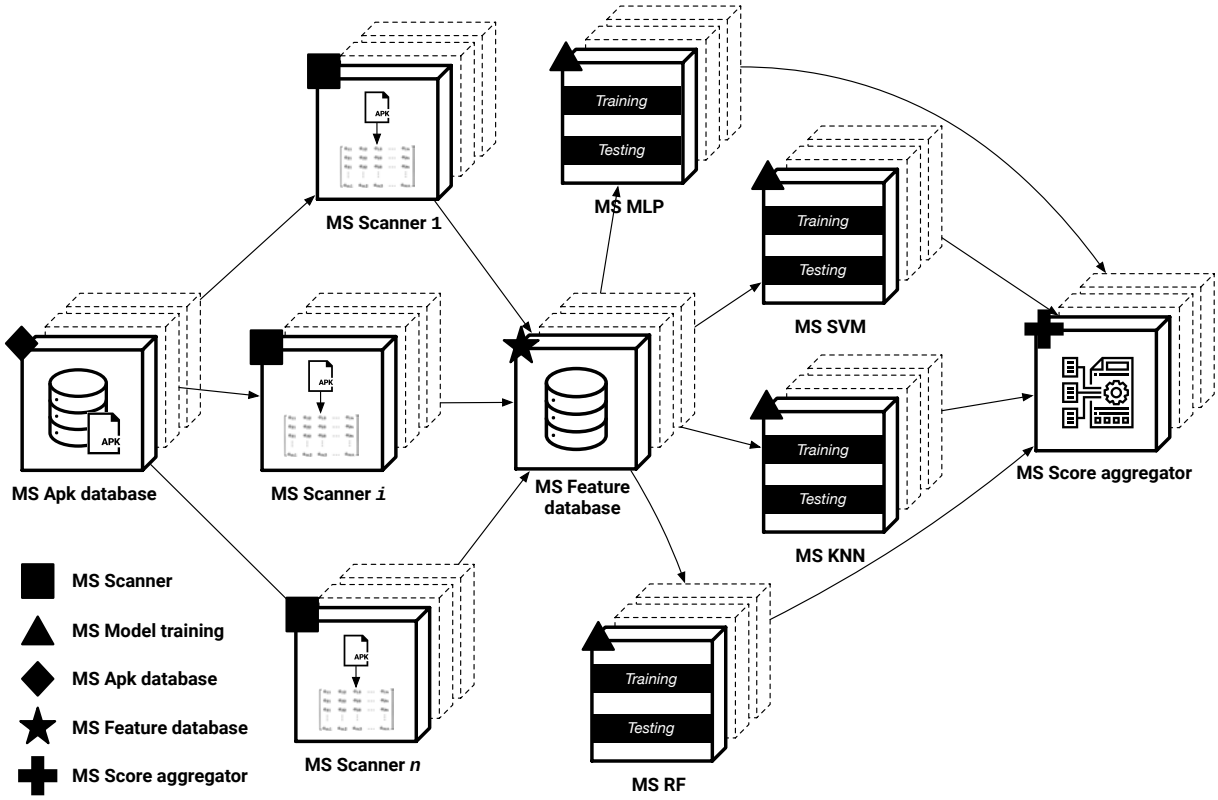

Figure 3.2 – Overview of DroidAutoML

vices leverage *Scikit-learn* to perform both training, and testing steps. Afterwards, each ▲ instance parametrizes its ML algorithm according to the input hyper-parameter combination. All ML models are trained with a 10-cross fold validation process to avoid *overfitting* problems. The input data is split according to machine learning ratio standards: 60% of the data is used to fit the model and 40% to test it. Performances of each model are assessed in terms of *accuracy* and *F1 score*. Finally, trained models are stored within the database along with the configured hyper-parameter settings so that they can further be used by the end-user. The obtained results on the testing set are then communicated to *score aggregator* microservices (see Figure 3.2,✚).

Microservices dedicated to score aggregation. A third set of microservices are the ones dedicated to the collecting of results from ▲ microservices to identify the pair *{algorithm,hyper-parameters}* that gives the best performances for a given scanner. Each score aggregator microservice is dedicated to a couple *{scanner, algorithm}* so that it collects only results related to it for all hyper-parameter combinations tested. Accordingly, for n scanners and l algorithms, there will be at least $n*l$ instances of aggregators. Once the best predictive model have been found for a given scanner, the corresponding algorithm and hyper-parameters are communicated to the end-user.

Efficient microservice scheduling. DroidAutoML is a system designed to run on top of a cluster of hardware machines. To optimize resources and efficiently schedule tasks on such a cluster, DroidAutoML leverages on a bin packing algorithm [156]. As such, by splitting scanner benchmarking operations into smaller tasks, DroidAutoML can capitalize on properties offered by microservice architectures. Firstly, DroidAutoML fully takes advantage of multi node clusters as each microservice can be scheduled independently on any node in the cluster. Secondly, as scanner benchmarks are parallelized, ▲ microservices can run side by side with ■ microservices as long as they do not work for the same scanner. Thirdly, if a microservice fails during its execution, only its workload is lost and it can be automatically rescheduled.

Implementation. DroidAutoML is built on *Nomad*, an open-source workload orchestrator developed by *HashiCorp* [157], which provides a flexible environment to deploy applications on top of an abstracted infrastructure. More precisely our Nomad instance federates a cluster of 6 nodes (see Figure 3.3, ❶) that accounts for 600GB of RAM and

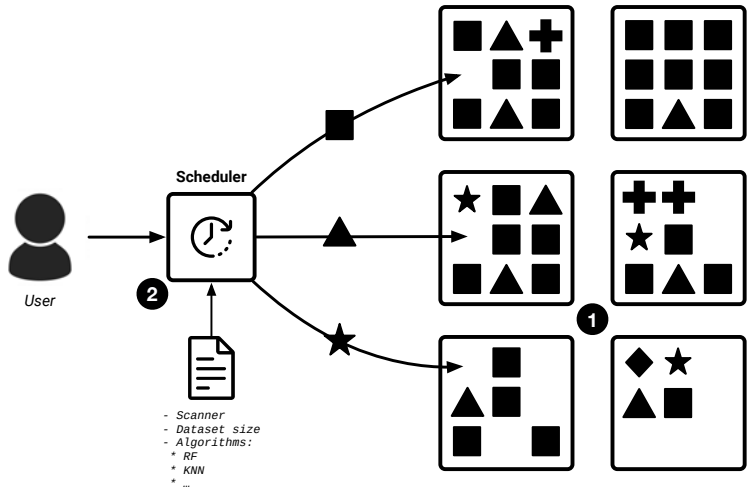

Figure 3.3 – Overview of DroidAutoML implementation

124 cores at 3.0Ghz. We use the bin packing algorithm implemented in *Nomad* to schedule (see Figure 3.3, ❷) DroidAutoML microservices instances across available nodes in the cluster as schematized in Figure 3.3.

Each microservice instance is represented as a *job* managed by the *Nomad* scheduler. Hardware resources allocated to each microservice depend on its type: scanner specific instances take 2 cores and 4GB of RAM each, model training instances take 1 core and 2GB of RAM, and score aggregator instances take 1 CPU and 1 GB or RAM. The time required for a scanner instance to build a feature vector depends on the size of the input apk as well as its operating time. It ranges from 6 seconds for a 2MB application to 61 seconds for a 100MB application on average. The apk database of DroidAutoML is currently composed of 11561 applications, 5285 malware and 6276 benign applications and the average size of an application is 20.25 MB with a standard deviation of 21.48.

Given the resources required for one instance, our infrastructure can run 61 ■ microservice instances in parallel, therefore the entire apk database can be processed in 24 minutes with our current cluster. The time required to train and test a ML model depends on the algorithm, the set *hyper-parameters* used, and the size of the input vector matrix. We provide in table 3.3 the minimum, average and maximum time required to train and test a model according to an algorithm. As we use a grid-search approach to perform *hyper-parameter tuning*, the number of ML models train to evaluate a scanner depends on the number of hyper-parameter combinations to test. The table 3.3 summarizes the values tested for each *hyper-parameter* according to an algorithm as well as the number of

combinations to test them all. For example, given the resource constraints of a ML model microservice, our cluster can run 123 ▲ microservice instances in parallel, thus testing all 3120 hyper-parameter combinations for the Random Forest takes on average 9 minutes for an input feature vector matrix of 11561 items.

3.4.3 DroidAutoML evaluation

To evaluate our approach, we propose to apply our microservice architecture to two state-of-the-art machine learning based malware detection systems in order to improve learning algorithm selection and training. More precisely, we conduct our experiments on approaches proposed by Drebin [21] and MaMaDroid [19]. We use the implementations provided by authors for both Drebin [1] and MaMaDroid [2] We benchmark our approach against the ground truth of the related work by reusing the same ML algorithms used by the two approaches: Support Vector Machine (SVM) for Drebin and Random Forest, SVM and K-Nearest Neighbors for MaMaDroid.

We build a dataset of 11561 applications composed of 5285 benign and 6276 malware samples. Malicious samples are collected from three malware datasets: the Drebin dataset [21], the `Contagio` dataset [88] and a dataset of 200 verified ransomware from `Androzoo` [158]. Concerning benign applications, we collect samples from the top 200 of most downloaded applications for each app category in the Google Play Store. To ensure that collected samples are really benign, we upload them to VirusTotal, an online platform that makes it possible to have a file analyzed by more than 60 commercial antivirus products. According to the literature [159], applications can be safely labeled as benign if less than 5 antivirus detect it as malware, as several antivirus consider *adwares* as potentially dangerous. Among the 6276 applications downloaded, $95,04\%$ (5965 samples) have not been detected as malware at all and $99,18\%$ (6225 samples) by less than 5 antivirus. To guarantee the overall dataset quality, we remove all samples with a detection rate over this threshold.

Ground truth results. As original experiments by Drebin and MaMaDroid authors were made on older data, both approaches may suffer from *temporal bias* [160, 161]. *Temporal bias* refers to inconsistent machine learning evaluations with data that do not correctly represent the reality over time. To take this bias into account, we start our exper-

1. Drebin source code: https://www.dropbox.com/s/ztthwf6ub4mxxc9/feature-extractor.tar.gz
2. MaMaDroid source code: https://bitbucket.org/gianluca_students/mamadroid_code/src/master

Scanner	Algorithm	Accuracy	F1-Score	Precision	Recall	TP	TN	FP	FN
Drebin	**SVM**	88.91	88.23	84.43	92.39	1833	2087	338	151
MaMaDroid	**KNN**	82.35	81.76	83.25	80.33	1744	1887	427	351
	Random Forest	80.54	83.08	72.65	97.01	2106	1445	65	793
	SVM	79.22	81.97	71.57	95.90	2082	1411	89	827

Table 3.1 – Baseline results for DREBIN and MAMADROID models trained with original *hyper-parameters* settings.

iment by measuring ground truth results for both DREBIN and MAMADROID approaches using our own dataset. These results will serve as a baseline to evaluate DROIDAUTOML performances and compare further results against it. Authors from DREBIN use a SVM algorithm to perform the binary classification of malware and benign applications. As the original source code of their approach is not available, we develop our own implementation of their solution using available information in the original paper. While our implementation of DREBIN may slightly differ from the original one, the approach and the algorithm used (SVM) remain conceptually the same. As no details are given about *hyper-parameters* used to parametrize the algorithm, we take common default values suggested by machine learning frameworks to train the algorithm. Regarding MAMADROID, authors tested three learning algorithms: Random Forest, SVM and KNN. We calculate the baseline by using the MAMADROID's approach source code, and the same *hyper-parameters* set by the authors. The table 3.2 reports the grid of *hyper-parameter* values

	Parameters	Mamadroid	Drebin
Random Forest	n_estimators	101	
	max_depth	32	
	min_samples_split	2	
	min_samples_leaf	1	
	max_features	auto	
SVM	C	1	1
	kernel	rbf	linear
	degree	3	3
	gamma	auto	auto
KNN	n_neighbors	[1,3]	
	weights	uniform	
	leaf_size	30	
	p	2	

Table 3.2 – Default hyper-parameters used to parametrize evaluated algorithms

used to train and test each learning models for both approaches. The table 3.1 reports the baseline results for each trained model. We observe that the accuracy and F1 scores

	Parameters	Hyper-parameters	# of combinations to test	time for a single run (in seconds for 11238 apks)		
				min	avg	max
Random Forest	n_estimators max_depth min_samples_split min_samples_leaf max_features	[200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, 2000] [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 500, 1000,None] [2, 4, 6, 10] [2,5,10,20] [auto,sqrt]	10*13*3*4*2=3120	15	21	35
SVM	C kernel gamma	[0.0001,0.001,0.01,0.1,1,10,100,1000,10000] [linear,rbf,sigmoid,poly] [0.0001,0.001,0.01,0.1,1,auto,scale]	9*4*7=252	23	25	31
KNN	n_neighbors weights leaf_size p	[1,3,4,5,6,7,8,9,10] uniform,distance [1,3,5,10,20,30,50,100] [1,2]	9*2*8*2=288	23	42	56
MLP	hidden_layer_sizes activation solver alpha learning_rate	[(50, 50, 50), (50, 100, 50), (100,)] [tanh, relu] [sgd, adam, lbfgs] [0.0001, 0.05] [constant, adaptative]	3*2*3*2*2=72	123	164	250

Table 3.3 – Grid hyper-parameters used to train models with DroidAutoML

for both approaches decrease compared to the original results. The accuracy score for the Drebin SVM drops by 5.09% from 94% to 88.91%. Considering MaMaDroid, F1-Scores are below 84% for all studied algorithms, with a false-positive rate over 5%, which is more than 15% lower than best results presented originally in terms of F1-Score. As samples in our dataset are more recent than those used in original experiments, these results confirm that both Drebin and MaMaDroid approaches are suffering from *temporal bias*.

Model evaluation with DroidAutoML. In the following of this experiment, we aim at answering the following questions:

— **RQ1**: Is DroidAutoML able to find a learning algorithm that performs better than default algorithms used for studied scanners?

— **RQ2**: Can DroidAutoML improve the prediction results of studied scanners by finding an optimal set of *hyper-parameters* ?

We answer these questions by running DroidAutoML for each studied scanner with a large grid of *hyper-parameters* (see Table 3.3) and 4 different learning algorithms for each scanner: Random Forest, SVM, KNN, and a multi layer perceptron (Neural Network).

The table 3.4 reports the best results obtained for both Drebin and MaMaDroid. For the Drebin approach, accuracy and F1 scores of the model trained with SVM increase by 10.59% and 11.27% respectively compared to the baseline. Moreover, we observe that the multi layer perceptron algorithm performs slightly better than the SVM algorithm with +0.11% in accuracy and +0.10% in F1-Score, thus reducing the number of false negative from 19 to 10. DroidAutoML also succeeds to improve MaMaDroid baseline results for all three studied algorithms. In details, DroidAutoML increases

Scanner	Algorithm	Accuracy	F1-Score	Precision	Recall	TP	TN	FP	FN
Drebin	**KNN**	98.82	98.82	99.91	97.75	2169	2188	2	50
	Random Forest	98.57	98.56	99.63	97.52	2163	2183	8	55
	SVM	99.50	99.50	99.86	99.13	2168	2219	3	19
	MLP	99.61	99.60	99.68	99.54	2164	2228	7	10
MaMaDroid	**KNN**	85.48	86.41	93.69	80.17	2034	1735	137	503
	Random Forest	87.93	88.57	94.98	82.98	2062	1815	109	423
	SVM	88.97	88.49	86.09	91.03	1869	2054	302	184
	MLP	84.71	85.36	90.55	80.73	1966	1769	205	469

Table 3.4 – Best results after model training on DroidAutoML

	Parameters	Mamadroid	Drebin
Random Forest	n_estimators	1600	1500
	max_depth	50	30
	min_samples_split	2	4
	min_samples_leaf	2	10
	max_features	sqrt	auto
SVM	C	1000	1000
	kernel	linear	rbf
	degree	3	3
	gamma	auto	scale
KNN	n_neighbors	6	5
	weights	uniform	uniform
	leaf_size	50	20
	p	2	2
MLP	hidden_layer_sizes	100	50,100,50
	activation	tanh	tanh
	solver	adam	lbfgs
	alpha	0.05	0,0001
	learning_rate	adaptative	constant

Table 3.5 – Hyper-parameters found for best case performance

MaMaDroid's SVM baseline accuracy by 9.75%, KNN by 3.13% and RF by 7.39%. These accuracy improvements are accompanied by a significant increase of F1-scores for all algorithms. It represents a significant decrease of the number of false positives and false negatives. In their paper, MaMaDroid's authors discard the SVM algorithm due to poor performance compared to other algorithms tested. We show here that SVM is actually better than other algorithms tested by authors when it is parametrized with the adequate *hyper-parameter* values as shown in table 3.5. Notice that in machine learning, optimal *hyper-parameters* values depends on the problem to solve [145]. Therefore, as the feature vectors are encoded differently for Drebin and MaMaDroid, optimal *hyper-parameter* values may slightly differ from one approach to the other.

We answer **RQ1** by showing that DroidAutoML has been able to find a ML al-

gorithm that performs better than those tested empirically with studied scanners. More precisely, the Multi Layer Perceptron outperforms the SVM algorithm used by DREBIN originally and the MAMADROID SVM originally discarded by the authors due to poor results performs better than other algorithms initially retained (i.e. RF and KNN).

Furthermore, we answer **RQ2** by showing that DROIDAUTOML has been able to find a combination of *hyper-parameters* in a reasonable amount of time (less than 30 minutes) that enables to significantly improve prediction results for all machine learning models trained for studied scanners.

3.4.4 Groom: A feature engineering framework to reach more efficient ML malware detection models

Like parameter optimization, feature engineering is an essential aspect of classification problems in machine learning. Feature selection, extraction and encoding are essential steps towards building efficient ML based malware scanners. Features must be chosen in such a way that ML algorithms can easily differentiate studied classes in the problem and thus succeed in generalizing the problem. When poorly chosen, algorithms may be unable to generalize the problem or suffer from *overfitting*.

Type	Features	Static analysis	Dynamic analysis	Location
Structural	permissions	✓	✗	manifest
	intent-filters	✓	✗	manifest
	components	✓	✗	manifest
	file signatures	✓	✗	apk level
	protected method calls	✓	✓	bytecode
	suspicious method calls	✓	✓	bytecode
Behavioural	call graph	✓	✓	bytecode
	dynamic code loading	✗	✓	apk level
	network traffic	✗	✓	OS level
	intent messages	✗	✓	OS level

Table 3.6 – Examples of structural and behavioral features that can be extracted from an Android application

Manual feature selection. Feature selection consists in automatically or manually selecting features which contributes the most to correctly classify your incoming data. Having irrelevant features can decrease the accuracy of the model and makes it unable to grasp the difference between benign and malicious applications. In the Android ecosystem, several feature selection processes have been explored in the recent years [19–21]. Mainly, we distinguish two types of features: (i) structural and (ii) behavioural features, as

illustrated in Table 3.6. Structural features are pieces of information inherent to an application, but which by themselves do not directly encode its corresponding behavior [21]. It is the correlation of those structural features altogether that allows the model to perform classification. For instance, the permission `READ_CONTACTS` can be used by both benign and malicious applications and does not give much information about intentions of the application. However, when correlated with the API call `url.openConnection()`, it may highlight the intentions of a malicious application to steal user's contacts. Contrariwise, behavioural features are information about an application that allows to capture both intentions and actions of an application [19]. Behavioural features may be meaningless unless they are interpreted and encoded correctly to reveal a particular behaviour. For example, this can be done by encoding sequence of calls from the application call graph [19] or by encoding the network trace of an application during its execution [162].

Feature encoding. Feature vectors are used to represent the characteristics of studied samples numerically to simplify their analysis and comparison. Most of ML classification algorithms such as (e.g. Random forest, Support Vector Machine, etc.) use feature vectors as input to train their model. While it is easy to use pixels of an image as a numerical feature vector, it is harder to numerically encode more complex features such as basic or behavioral features of Android applications. For that reason, many studies [19–21, 163] provide new alternatives for feature encoding. For example, authors of Drebin [21] embed structural extracted features into a feature vector using *one-hot encoding* to code the presence, or the absence of a given feature. Contrariwise, MaMaDroid [19] encodes *behavioral* extracted features using a Markov chain algorithm, which calculates the probability for a method to be called by another in a call graph.

Limits of feature engineering approaches in existing studies

For many binary classification problems in the field of machine learning, one of the biggest challenge resides in the model's ability to minimise the number of false positives, i.e legitimate applications labeled as malware and false negatives, i.e. malware labeled as benign applications. Regarding Android malware detection, a high number of false positives can negatively impact the business of companies, as legitimate applications will be wrongly discarded from public market places. In a similar way, a high number of false negative presents an important security issue because harmful applications can mistakenly reach public market places and harm end users.

Limits of features used in state-of-the-art ML-based scanners. The set of features extracted from applications and used to train the classification model can greatly influence the performances of the model. As such, studies leveraging machine learning to perform malware detection [19–21, 81, 164] explored various approaches regarding feature engineering. Previous works [165] used to rely on dangerous permissions requested by the application to build learning models. However, this strategy is prone to false positive since many benign applications make use of these dangerous permissions for legitimate reasons [166].

Later, authors of DroidAPIMiner [20] build a model relying on the frequency of API system calls to build their classification model. But such an approach need a constant retraining given the rapid evolution of malware as well as the Android API. To counter this phenomenon, authors of MaMaDroid [19] proposed a novel approach which relies on sequence of abstracted API calls to build a model more robust over time. Unfortunately, the model proposed by MaMaDroid has proven to be less robust to malware evolution than claimed originally [39]. This is because its feature engineering approach may capture relations in the training data that quickly become obsolete at test time to differentiate malware from benign applications.

In 2014, authors of Drebin [21], proposed an approach based on a broad static analysis to gather as many features as possible from the application's bytecode and manifest, such as suspicious API calls, permissions or application's component names. Features are then organised as strings in a joint vector space which allows Drebin to identify combinations of features that may represent a malicious behaviour. However, the Drebin static analysis neglects features that can help the model to detect malware using complex evasion techniques such as dynamic code loading or reflection, that allow attackers to evade static analysis. Such limitations emphasizes the need for more training classification models with more sets of features that are more robust to the evolution of malware over time.

Badly chosen features can introduce overfitting in classification. The Drebin [21] approach, for existing implementations available publicly, has a severe drawback regarding feature encoding. In the original paper, authors stipulate that a malicious behaviour can be expressed as specific patterns and combinations of extracted features. For example, a malware stealing user contacts may call the API method `getContacts()` and ask the `PERMISSION.INTERNET` permission. To capture the dependency between features, they

map extracted features represented as strings to a numerical joint vector space. To do so, they first constitute a set that comprises all strings observed in samples from the training dataset. Then, using this set, they define a $|S| - dimensional$ vector space where each dimension value is binary: 1 if the application hold a given feature and 0 otherwise.

Consequently, the approach rely on a set of feature observed a time T that may not be accurate at a time $T+1$ for making prediction on real data. For example, authors include application component names (Activities, Services, etc.) into the feature set. While such features can be very accurate to detect malware variants with the same component names, as soon as an attacker creates a new variant with different component names, the model become obsolete and need to be retrained with new samples. The model will only ever know about features from the training set, hence if features are too discriminative, such as application package name or unique strings, the model could not be able to generalize for unforeseen malware variants at prediction time. Therefore, the current implementation of DREBIN needs a constant retraining to stay up to date and will be unlikely to catch 0-day malware.

This issue suggests that the set of features chosen to train machine learning models must be carefully chosen *a priori* in order to build machine learning models that remain robust over time. For example, with MAMADROID [19], authors build a fixed-length feature vector by abstracting the sequence of API calls in the application call graph using java package names instead of the full method name. Similarly, with DROIDAPIMINER [20], authors carry out a deep analysis of malware behaviours to build a finite feature set composed of 8375 distinct API calls. Choosing a fixed set of features to to be extracted from an application appears to be a safe and reliable approach to build robust and re-usable machine learning models.

3.4.5 Groom approach

In this work, we propose an original approach that leverages on a broad static analysis to improve the quality of features extracted from an application compared to the DREBIN [21] original paper. Our tool, which we call GROOM, leverages on the Soot framework [167] to quickly analyse the application's bytecode, the application's manifest and application's resources. In particular, GROOM extracts specific Android API calls that enable to take into account the use of recent obfuscation techniques such as reflection, native code execution or dynamic code loading. In terms of resource consumption, GROOM is less expensive than previous approaches (DREBIN and MAMADROID) while beign more

Features	**Drebin**	**Groom**	**Location**
Permission	✓	475	Manifest
Hardware components	✓	136	Manifest
Component names	✓	✗	Manifest
Component count	✗	✓	Manifest
Intent filters	✓	789	Manifest
Restricted API calls	1266	4942	Bytecode
Used permissions	✓	✓	Bytecode
Suspicious API calls	120	230	Bytecode
Strings	URLs only	✗	Bytecode
String type count	✗	✓	Bytecode
Sources	✗	18077	Bytecode
Sinks	✗	8324	Bytecode
Abis	✗	7	APK

Table 3.7 – Details of features extracted by Drebin and Groom during static analysis

efficient. To choose what features Groom should extract, we rely on several key studies in the field of malware detection and Android vulnerabilities [20, 45, 46, 51]. Extracted features can roughly be categorized on two subsets: (i) bytecode related features and (ii) resource related features. Bytecode features are derived from the decompilation of the apk `dex` file(s) whereas resource related features are features derived from application resources and its manifest file. The table 3.7 shows a comparative overview between features used by Groom and Drebin. Notice that the implementation of Drebin to which we compare is more recent than the one used for the original paper and has been used in recently published papers [168]. As such, features that may evolve over time such as permissions or API calls has been updated to fit with latest Android versions.

Groom feature selection. We now describe in more depth how we carried out feature selection for Groom and how it compares to Drebin. Firstly, Groom exploits several types of features supported by Drebin but bring improvement to them. As such, Groom greatly increases the number of restricted API calls monitored during static analysis from 1266 to 4942. Restricted api calls describes Java method calls that are protected by an Android permission. In order to increase the number of monitored restricted calls, we wrote a script to parse the entire Android documentation base as well as XML manifests in the Android sdk repository to find new links between permissions and method calls. Groom also increases the number of suspicious API calls up to 450 instead of 120

for DREBIN. While several families of suspicious api calls overlaps between DREBIN and GROOM (such as crypto, network or native code loading related calls), GROOM considers new families such as reflection and dynamic code loading related methods. Contrary to DREBIN which extracts all networks URLs string from the apk, GROOM creates features by counting occurrences of specific types of strings in the application. GROOM counts the occurrences of phone numbers, encrypted strings, method names passed as argument, network urls, and so on. Such technique allows GROOM to encode suspicious behaviours based on string patterns, such as method arguments, that can for example used to invoke methods suspicious through reflection. To successfully extract these strings, GROOM implements an Inter-Component and backward analysis on the application call graph that enables to track string variable in the call graph up to their definition.

In complement to features extracted by both approaches, GROOM adds 4 new sets of features. We leverage the work SuSi [169] to identify sources and sinks used by the application. A source represents an Android API call where the application can collect privacy sensitive data such as `getLastKnownLocation()` that allows to get the gps location of the device. A Sink represents an Android API call that allow the application to leak private data through a channel to an adversary. Finally, to detect the use of native libraries, GROOM extracts the CPU instruction sets that the application support (`abis` [170]).

Groom feature encoding. Similarly to DREBIN, GROOM encodes all extracted features in a joint vector space. The absence or presence of a particular attribute, such as a permission, is encoded as a binary feature. Numerical properties such as the number of *Activities* are encoded as continuous features. But contrary to DREBIN, the final set of extracted features of GROOM is finite and does not depend on the size of the training set. GROOM extracts only features that can be shared by several applications. GROOM has a fixed set of features (of size 33077) which results in a fixed feature vector length for any application analyzed. DREBIN extracts features that are potentially unique for a given application, such as component names or network urls, the size of the DREBIN vector depends on the number of unique features found when analyzing all samples from the training set. Thanks to this models trained with the GROOM approach are faster to train and do not need to be constantly retrained to make accurate predictions. As soon new unknown applications must be tested, GROOM can analyse them and build a feature vector that will be compatible with a model trained previously.

3.4.6 Groom evaluation

For this experiment, we implement a machine learning pipeline for GROOM and add it to DROIDAUTOML (see section 3.4.1), our scalable architecture that allow to compare ML based Android malware detection approaches. We begin our evaluation by describing our experimental setup and evaluation metrics. We then address the following questions:

— How GROOM performs compared to state-of-the-art scanners approaches DREBIN [21] and MaMaDroid [19] ?

— Is GROOM less expensive than DREBIN and MaMaDroid in terms of computational resources ?

Experimental setup. We evaluate GROOM as a binary classifier, that said, its capacity to correctly classify benign and malicious samples from a dataset. We compare GROOM to two state-of-the-art machine learning malware detection approaches, DREBIN [21] and MaMaDroid [19]. To correctly compare GROOM to the state-of-the-art, we rigorously apply the same experimental settings for all three approaches. As such, all approaches are tested with the same dataset, the same learning algorithms and the same hyper-parameters settings. To evaluate each approach in best possible performances, we leverage DROIDAUTOML (see section 3.4.1). Thanks to DROIDAUTOML, each classifier can be evaluated with a large set of hyper-parameters on three algorithms: Random Forest (RF), Support Vector Machine (SVM) and K-Nearest Neighbors (KNN). Notice that in the original paper, DREBIN authors only test their approach using an SVM.

For evaluating the classification results, we compare each approach using the *accuracy* score, the *F1-Score* score, the number of false positives and the number of false negative of each classifier. Notice that the F1-Score is calculated from the *precision* and *recall* scores of the classifier that are both quality metrics used in the field of machine learning. To avoid *overfitting*, we follow the guidelines of the machine learning community. All classifiers are trained with a 10-cross fold validation and we perform a hold-out validation: samples used to train the model are different from those used to validate it. As such, 60% of samples are used to train the model and 40% to validate it.

To carry out our experiments, we first generate a balanced dataset D_1 composed of 50610 samples: 25305 benign applications and 25305 malware. To collect samples, we leverage the Androzoo database [158]. We collect benign and malicious found in the wild between 2017 and 2020. We consider a sample as benign if exactly no antivirus among 60 on the VIRUSTOTAL platform detected the sample as malicious. Contrariwise, we consider

a sample as malicious if more than 10 antivirus detected it as malicious. With TESSERACT, authors suggest that the proportion of malware in the dataset can affect the performances of the classifier. Therefore we follow authors's recommendation and build a dataset D_2 by downsampling the number of malware in the datasets D_1. To fit these proportions in our dataset D_2, we sample the datasets D_1 by keeping 100% of benign applications and randomly picking 10% of malware applications. The final dataset is composed of 30000 samples, 25000 benign and 5000 malicious applications.

Approach	Algorithm	Accuracy	Precision	Recall	F1-Score	FN	FP	TN	TP	Train size	Test size
Drebin	**KNN**	94.90	96.45	93.22	94.81	686	347	9775	9436	30366	20244
	RF	95.64	95.54	95.76	95.65	429	453	9669	9693	30366	20244
	SVM	95.71	96.44	94.93	95.68	513	355	9767	9609	30366	20244
Groom	**KNN**	97.39	97.83	96.94	97.38	310	218	9904	9812	30366	20244
	RF	97.28	98.25	96.28	97.25	377	174	9948	9745	30366	20244
	SVM	96.80	96.86	96.75	96.80	329	318	9804	9793	30366	20244
mamadroid	**KNN**	92.29	91.94	92.72	92.33	737	823	9299	9385	30366	20244
	RF	92.94	91.78	94.32	93.03	575	855	9267	9547	30366	20244
	SVM	87.59	84.75	91.66	88.07	844	1669	8453	9278	30366	20244

Table 3.8 – Best results obtained by Groom, DREBIN and MaMaDroid for each learning algorithm tested

Experimental results. We now present our experimental evaluation of GROOM. Using the dataset D_1 summarized in paragraph 3.4.6, we measure the accuracy of GROOM's classification on benign and malicious samples and compare it to DREBIN and MaMaDroid. Results are reported in table 3.8. As shown in table 3.8, GROOM models obtain an accuracy and a F1-Score superior to all models from DREBIN and MaMaDroid. The SVM algorithm achieves the best results for the DREBIN approach with an accuracy of 95.71% and a F1 score of 95.68%. For MaMaDroid, the Random Forest algorithm obtains the best results with an accuracy score of 92.95% and a F1 score of 93.03%. Comparatively, the KNN algorithm with GROOM outperforms DREBIN and MaMaDroid's best algorithm with an accuracy scor of 97.38% and a F1 score of 97.25%. As such, GROOM obtains an accuracy score that is 1.67 percentage point better than DREBIN's SVM and 4.43 percentage point better than MaMaDroid's RF in the same experimental settings.

We now repeat a similar experiment by changing the number of malware and benign applications in the dataset to make proportion closer to reality. To do so, we measure the accuracy classification of the three approaches using the dataset D_2. Results obtained are reported in table 3.9. Similarly to experiment with dataset D_1, models trained with the

Approach	Algorithm	Accuracy	Precision	Recall	F1-Score	FN	FP	TN	TP	Train size	Test size
Drebin	**KNN**	98.78	96.20	90.12	93.06	100	36	10086	912	16701	11134
	RF	98.86	98.57	88.74	93.40	114	13	10109	898	16701	11134
	SVM	98.78	95.06	91.30	93.14	88	48	10074	924	16701	11134
Groom	**KNN**	99.04	97.78	91.50	94.54	86	21	10101	926	16701	11134
	RF	98.98	98.38	90.22	94.12	99	15	10107	913	16701	11134
	SVM	99.03	95.94	93.28	94.59	68	40	10082	944	16701	11134
mamadroid	**KNN**	95.79	77.62	75.40	76.49	249	220	9902	763	16701	11134
	RF	95.72	84.45	64.92	73.41	355	121	10001	657	16701	11134
	SVM	90.91	0.00	0.00	0.00	1012	0	10122	0	16701	11134

Table 3.9 – Best results obtained by Groom, Drebin and MaMaDroid on an unbalanced dataset

Groom approach outperform other models trained with both Drebin and MaMaDroid. The Groom model trained with the KNN algorithm obtains the best results with an accuracy score of 99.04% and a F1 score of 94.54%. The best performances for Drebin are achieved with the RF algorithm with an accuracy score of 98.86% and a F1 score of 93.40% which is 0.18 and 1.14 percentage point lower than the best Groom's model. Finally, KNN is the best model for MaMaDroid, with an accuracy score of 95.79% (-3.25 compared to Groom) and a F1 score of 76.49% (-18.05 compared to Groom).

Performances and computational resources required to perform static analysis heavily depends on the size of the application analyzed. For the same amount of CPU and RAM allocated and given the same application, Groom is on average 1.5 times faster than Drebin and 5 times faster than MaMaDroid. This is due to the fact that Groom makes use of the Soot framework [167] and FlowDroid [51] which are optimized to decompile Java/Dex bytecode. Moreover, contrarily to MaMaDroid, Groom does not need to extract the application call graph neither to make calculation on it.

3.5 Conclusion

In this chapter, we have shown that performances existing machine learning based scanners can be greatly improved using hyper-parameter optimization and algorithm selection. We see that, in a reasonable amount of time and inexpensive resources, our microservice architecture, DroidAutoML, can systematically find an optimal set of hyper-parameters that allow to significantly improve all test learning algorithms. Besides, we presented Groom, a novel static analysis approach to efficiently extract a fixed set of features, adequately chosen to discriminate Besides, whit Groom, we presented a new

set Besides with Groom, we present a new sets of features, adequately chosen and extracted with static analysis, that allow to greatly improve malware detection without any supplementary cost.

To evaluate both DroidAutoML and Groom, we used publicly available malware datasets, which are criticised for their poor quality. In the next chapter, we address the problem of unrealistic malware datasets: we study the creation of synthetic malware variants that use realistic obfuscation techniques observed in the wild. Crafting corner case malware variants will help to build challenging datasets for malware scanners.

Chapter 4

Reaching limits of antivirus leveraging the creation of experimental datasets

To evaluate precision and performances of antivirus scanners, reliable ground truth datasets is of the utmost importance. While previous studies [19–21] in the last decade reported promising results regarding malware detection on Android, most of them rely on small and outdated datasets such as the MalwareGenome dataset [7]. Unfortunately, it has been proven [39] that such datasets do not reflect the current fast evolving malware ecosystem. In this chapter, we propose a novel approach that aims to synthetically build malware variants with recent evasion techniques used by attackers in order to build challenging datasets for Android malware detection systems. We first discuss the inherent limitations of existing malware datasets in the Android research community. Then, we give background knowledge about complex obfuscation techniques used by malware to evade detection. Finally, we present KillerDroid, a toolchain that enables to generate new malware variants using complex obfuscation techniques to build challenging malware datasets.

4.1 Evolution of Android malware datasets

Recent studies [23, 39] pointed out the inherent limitations of datasets widely used datasets to train ML-based malware detection models in the Android community. Most popular datasets [7, 21, 40] have been created six years ago and contain a limited number of samples. More importantly, many samples in these datasets are out of order or ineffective because they were designed to work on outdated Android operating system versions.

In this context, to better evaluate existing malware detection approaches, several studies [23, 33, 107, 110] proposed new approaches to generate more reliable Android malware

datasets. With RmvDroid [23], authors create new malware datasets by selecting applications that are both flagged malicious by VirusTotal and removed by Google from the Google Play Store. Other studies [33, 107, 110] propose to build synthetic malware datasets by automatically creating malware variants. These techniques make it possible to evaluate the robustness of existing malware detection scanners by generating highly obfuscated malware samples.

However, to the best of our knowledge, such studies apply only a small part of existing evasion techniques used by malware attackers to hide their malicious payload. Moreover, these studies do not address the combination of multiple obfuscation techniques together to build malware samples.

4.1.1 Obfuscation of Android applications.

Obfuscation is not reprehensible per se. It helps to protect the key algorithms and data structures in software from hackers. Unfortunately, obfuscation has been largely leveraged by malware authors to hide their malicious payloads from analysts' eyes. Experiments recorded in the literature [37] have further shown that using simple obfuscation techniques on old (i.e., commonly known) malware samples makes them hard to be detected by traditional antivirus systems, which often use signatures. A recent study [171], however, has shown that some latent features (e.g., instructions sequence) remain relatively unchanged no matter how morphed the samples become. This assumption has led to recent successes in machine learning-based malware detection. Unfortunately, normal obfuscations are easy to detect (through machine learning for example), and the detection rates may be inflated by the tendency of the approaches to model traits that are actually irrelevant to maliciousness (e.g., the presence of common library APIs used to implement piggybacking). As such, malware authors may use more complex obfuscation techniques that are ignored by machine learning models to successfully bypass detection.

4.1.2 Review of recent evasion techniques on Android

In the following of this section, we propose an overview of four major evasion techniques used by malware attackers to defeat existing scanners. These techniques are particularly troublesome because they are used both by benign applications as obfuscation techniques to protect their intellectual property and by malicious samples to dissimulate their malicious payload. For each obfuscation technique, we detail how both benign and malicious

samples are using them.

Repackaging. The repackaging technique consists of decompiling an Android application to make modifications on its bytecode, most often when the application source code is not accessible. Several off-the-shelf tools [167, 172, 173] exists to ease the decompilation process and translate the dalvik bytecode into an intermediate representation easier to manipulate.

Without malicious intentions, this solution can be used for several reasons. For example, several studies use application repackaging to enforce privacy rules [44, 121] or to remove application ads that monitor user private data [174]. In other cases, repackaging can be used to fix a bug within an application [175] or optimize application performances [176].

However, the repackaging technique has been diverted as a powerful attack to build malware [177]. To carry out this attack, attackers modify the content of a legitimate application by reverse-engineering it to write some malicious procedures directly in the legitimate bytecode. This technique allows an attacker to keep the overall look of the application while being able to execute its malicious code. The repackaging attack is in particular used by phishing attacks targeting applications such as online banking [178]. In such cases, the attacker tricks the user by repackaging a legitimate banking application to send the user's bank credentials to a remote server. In addition, the attacker may re-upload the modified application to a popular market to trick Android users [77]. This attack is particularly stealthy as it can be difficult to identify the difference between the legitimate application and the malicious updated one. In 2011, the `DroidDream` trojan has been found in more than 50 applications in the Android official market [179].

Packing. A packed application consists of a fake host Android application in which the attacker hides a genuine payload in a separate file and use several dissimulation techniques such as encryption and dynamic code loading to hide it from researchers and detection scanners. The host application can be an existing benign application, a blank application or any custom crafted application. Tools such as *Bangcle* [180] imitates the behaviour of malware packers to obfuscate legacy application for copyright purposes. In such cases, the original benign application bytecode is encrypted and packed into a blank application that can be uploaded on any market place. Yet, researchers know that this technique is widely used in the Android malware ecosystem [181, 182].

Reflection. Reflection in Java gives the ability to a program to manipulate objects repre-

senting the state of the program at runtime. While Android is based on the Dalvik virtual machine, the reflection API is almost the same as of Java. Generally, reflection is used to manipulate runtime information: access class data, create objects, invoke methods, change class field values,etc [183]. The reflection API is used for several legitimate usages:

1. **Backward compatibility**. The Android framework continue to encourage reflection usage because it is used for backward-compatibility between existing Android OS versions [184]. In this case, reflection is used to call API methods that have been marked as private or hidden (with the `@hide` annotation) in a previous Android SDK version.
2. **Access private methods**. The java compiler enforces some rules to prevent access to class, methods and field that hold the `private` modifier. However, the reflection API allows to manipulate these modifiers at runtime, hence giving the possibility to access private class members.

However, attackers use reflection to dissimulate their intention and bypass static analysis systems. A reflection attack leverages on the reflection API to create unexpected control flow paths within the application bytecode. The reflection attack is principally used to hide a malicious behaviour such as a suspicious method call from static analysis tools. Reflection allows to call any java method by passing the method's name as a string argument to the reflection api. That said static analysis tools must retrieve the argument passed to the reflection API to know which method will be called at a given point in the application call graph. Some static analysis based works tried to cope with java reflection by using intra and inter-procedural bytecode analysis to retrieve string arguments passed into reflection methods [185, 186]. However, these arguments may be hidden in other components within the application. For that reason these static analysis approaches are very expensive as they need to be context aware and they need to deal with ICC (inter component communication) [53]. ICC is the Android communication mechanism to communicate information such as strings from a component to another. Each Android component can broadcast intents to other components and can register receivers to receive intents from other components. For example, a developer can communicate a method name from an Activity to a Service and call this method using reflection within the Service. However, each intent sent from a component to another must pass upon the Android `Binder` service. It becomes therefore expensive for a static analysis tool to reconstruct all possible paths between components. Therefore it is difficult to perform a backward inter-procedural analysis to retrieve a string argument on Android. Finally,

	Benign	**Malicious**
Origin	Google Play Store	Drebin, Contagio, Androzoo
Dataset size	4491	5376
Reflection `java.lang.reflect`	 4464 (99.4 %)	 3400 (63.24 %)
Native code loading `java.lang.System`	 2780 (61.9 %)	 1077 (20.03 %)

Dynamic code loading
`java.lang.ClassLoader`
`dalvik.system`
	
	4262 (94.9 %)
	3502 (77.98 %)
		
		749 (13.93 %)
		28 (0.52 %)

Table 4.1 – Applications habits

it is possible to completely defeat these systems by encrypting argument string at compilation time and decrypt them only when they need to be passed to the reflection method.

Dynamic code loading Dynamic code loading is a functionality present in both Dalvik VM and Android runtime (ART) which allows a developer to load at runtime bytecode payloads from multiple locations (such as internal storage or over network). Our study on applications habits (see 4.1) shows that benign applications heavily rely on dynamic code loading (94% of benign applications use class loaders and 77% use dex class loaders) This functionality is used by benign applications for several reasons:

1. **dynamically update an application**. A developer can makes use of dynamic code loading to provide new features to an application without asking the user to update it through the Google Play Store. The functionality is also used to hot patch an application when a bug or a security issue is discovered. While it is discouraged by Google [187], three quarters (77%) of top 100 categories application from the Google Play Store use this feature (see 4.1).

2. **Circumvent the 64 method reference limit**. Only 64k methods can be referenced in a single dex file. To overcome this issue, developers can build applications with using several dex files (multidex), However, multidex significantly increases the application size on the Google Play Store and can discourage customers to download the app. To circumvent this limitation, developers upload small applications with a single dex file to the application market and dynamically load remotely the missing dex files gradually when needed.

3. **Hide sensitive code**. Dynamic code loading can be used along with encryption to hide bytecode payloads. This technique can be leveraged to obfuscate an ap-

plication to makes it difficult to reverse-engineer with static and dynamic analysis tools. Several tools [180, 188, 189] provide this feature to protect the intellectual property of companies.

For Android malware, dynamic code loading is a powerful technique to hide malicious bytecode from static analysis tool. For example, an attacker could build a malware that pretend to be a simple news application. When uploading it to Google Bouncer, the static analyzer would not notice any suspicious behavior because the application only expose a benign control flow. However, when a user will download and execute the application, this one will trigger a dynamic code loading procedure to remotely load the malicious code and perform dangerous actions. In our study we show that the number of malware that loads code dynamically is not as numerous as one might think (only 28 samples of a dataset of 5376 applications 4.1). However, there may be more than we think for two reasons. Firstly, some malicious applications may have used unknown techniques to dynamically load their malicious payload, thus bypassing our static analysis technique. Secondly, dynamic code loading is a more recent attack that may not be implemented in available malware datasets. This issue stresses even more the importance of training android malware detection systems with recent datasets that represent the current malware reality. Considering the difficulty to detect this attack using only static analysis, several studies [190, 191] leveraged on dynamic analysis to catch dynamically loaded code.

Bytecode encryption. Bytecode encryption is a technique using along with *dynamic code loading* which consists of encrypting some bytecode and only decrypting it when dynamically loaded at runtime. For intellectual property reasons, Android actors are sometimes encrypting the third-party libraries they provide to developers to develop their application. In such cases, the developer add a small amount of code in their app which takes care of loading and decrypting the third party library code at runtime. Typically, encrypted binary code is decrypted using the `javax.crypto` package which is built-in in the Android framework.

However, the `javax.crypto` package is also used by benign applications to encrypt *shared preferences* (i.e data stored locally and visible by all installed applications) or network requests. It is difficult to distinguish the different uses of this package using only static analysis methods. We are not aware of works that studied the amount of bytecode encryption in benign applications. However, it has been proven that bytecode encryption used along dynamic code loading has been used by malware to dissimulate their malicious

behaviors [192, 193].

Typically, an attacker will use a symmetric encryption algorithm such as `AES` to encrypt the file at compilation time. The key is stored in a field in the apk bytecode along with a decryption method that will take care of decrypting the binary file before being dynamically loaded by the class loader at runtime. In our study, we show that the `javax.crypto` package is used by almost all applications in the Google Play Store (89%) whereas it is not that common for malicious applications (33%) (see 4.1). Consequently, analyzing the use of this library does not allow to easily differentiate benign from malicious applications.

The study of these four major evasion techniques show how malware detection remains daunting task. Existing malware datasets lack malware variants implemented with such evasion techniques, thus preventing machine learning-based scanner to trained appropriately to detect them. Therefore, the observation made in this section emphasize the need to generate corner case malware variants, that leverage these evasion techniques to hide their malicious payload, in the aim of diversifying existing malware datasets.

4.2 KillerDroid: Weaving Malicious Payloads with Benign Carriers to Massively Diversify Experimental Malware Datasets

In this work, we present KillerDroid, a toolchain that enables to generate, in the dark, Android Adversarial Examples (AAEs) to evaluate the robustness against adversarial attacks of the state of the art malware scanners from both the academia and industry. Particularly, we consider that AAEs are generated in the dark as targeted malware scanners are seen as black box. Consequently, we consider that KillerDroid has absolutely no knowledge about both (i) the internal mechanisms/machine learning algorithms, and (ii) the datasets used to train targeted malware scanners. To produce valid AAEs while being blind, KillerDroid does not alter directly the bytecode of existing malware samples to guaranty their maliciousness. KillerDroid relies on well known studied piggybacking techniques [194] to dissimulate a valid malware payload inside a benign application. Consequently, perturbations are expressed in terms of code modifications over benign applications to silently inject, hook and hide a malware payload using a random combination of the latest obfuscation tactics. Then, the adversary sends the resulting piggybacked

application, considered thereafter as an AAE, to the targeted scanner, to search the adequate perturbations so that the produced AAE misleads the malware classification, and evades the detection to be finally flagged as benign. The altered benign application is considered as the attack vector that finally acts as a trojan. Indeed, once executed, the AAE switches the execution flow of the benign application to the malware payload as soon as the injected hook is reached.

4.2.1 Approach

To generate AAEs from existing malware, KillerDroid takes three inputs (See Figure 4.1, step ❶): (i) a benign application that will act as a *host*, (ii) a malware application considered as *guest* to be embedded into the *host*, and (iii) an obfuscation tactic expressed as a set of *obfuscations operations* to stealthily hide the guest malware into the benign application. KillerDroid leverages both Soot [167] and APKtool [172] to provide a decompiler (step ❷) that extracts all files from the APK given as inputs. Precisely, Soot is used to extract bytecode whereas APKtool is used to extract all other files such as resources (step ❸).

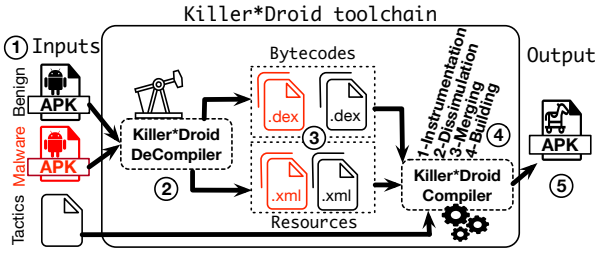

Figure 4.1 – Approach

Finally, from all the extracted files, the KillerDroid compiler applies a sequence of four phases: (i) instrumentation, (ii) dissimulation, (iii) resource merging, and (iv) APK building (steps ❹, ❺). Each phase is customized according to the obfuscation tactic given as input.

Instrumentation and dissimulation

The main aim of the instrumentation phase is to instrument the *host* to short circuit its legacy bytecode in such a way that at runtime it executes the malicious payload of the *guest* instead of its own code instructions (i.e., its bytecode). The short circuit must take

place as soon as possible to ensure the *host* does not execute any of its instructions. The purpose is to keep, as much as possible, the code instructions of the *host* intact, without executing it, in order to mislead scanners that are based on static analysis, while avoiding any interference at runtime with the malware behavior. Consequently, to guarantee that no *host* instructions are executed before the malicious payload, KillerDroid seeks the best place to inject the short circuit code sequence.

As depicted in Figure 4.2, as soon as an Android application is launched, its *ActivityThread* starts either `onCreate` (Figure 4.2, ❶) or `attachBaseContext` (Figure 4.2, ❷) method according to the way the application has been developed. Accordingly, KillerDroid instruments adequately one of this method by injecting a specific code sequence named thereafter the KillerDroid *bootstrap sequence* to instantiate at runtime the short circuit (Figure 4.2, ❸).

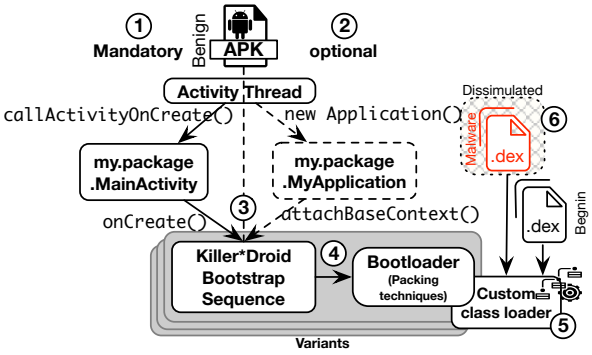

Figure 4.2 – Android application startup process

Bootstrap sequence purposes. The *bootstrap sequence* is a sequence of code instructions to correctly load, and execute the malware payload *via* the use of packing techniques [181, 182]. Precisely, this sequence enables: (i) to retrieve/decode the dissimulated malicious payload from the *host* APK, (ii) to decrypt the payload if encrypted, (iii) to save the obtained *guest* payload to the application local storage as a DEX file, (iv) to thereafter instantiate a custom classloader to load, from the previously saved DEX file, the malware classes instead of the host ones (Figure 4.2, ❺).

Avoiding scanners detection. One possible drawback of the injected *bootstrap sequence* is to become by itself a pattern characterizing the malicious intent of our approach, although packing techniques are not by themselves a malicious trait since they are commonly used by benign applications to protect intellectual property. If the same *bootstrap sequence* is injected repeatedly to thousands of benign applications to generate AAEs, the pattern

could be easily learned, and then recognized (e.g., by machine learning based scanners such as as a MaMaDroid [19], or Drebin [21]) as a feature that can be associated to malicious samples. We recall that our purpose is not to avoid unpacking mechanisms [195], but to avoid suspicions from scanners. Consequently, we leverage the concept of *generative programming* to generate thousands *bootstrap sequence* variants.

Bootstrap sequence variants. To craft unique bootstrap sequences while reducing suspicions, KillerDroid uses three obfuscation techniques for each generated AAEs: (i) fake call graph, (ii) methods renaming, and (iii) reflection. KillerDroid generates a call graph with a random depth, and spans randomly across the overall methods of the graph the code instructions of the bootstrap sequence. The methods' names are arbitrary (but yet plausible) names that come from two dictionaries of strings: one for prefixes, and another for suffixes. Both dictionaries have been extracted from the source code of the most popular Android projects on Github. KillerDroid is able to generate a pool of approximately 40 000 unique fake method names. To further increase the uniqueness of the sequence, each method call can be also invoked either directly in plain Java, or indirectly by using the reflection API or a mix of both (See Figure 4.3, ❶). As a result, machine learning based scanners will be challenged in finding call graph patterns to identify a malware. Furthermore, static analysis tools struggle to deal with reflection calls as they are not able to reconstruct a correct control flow [26, 196].

Nonetheless, even if KillerDroid is able to guarantee the uniqueness of the *bootstrap sequence* for each generated AAE, its injection directly into either the `onCreate` or `attachBaseContext` methods may still be considered as a suspicious pattern. For example, the use of specific methods, such as for example loading a `.DEX` file, is known to be a reliable suspicious pattern [190]. To overcome this potential issue, we have isolated the code related to packing techniques (i.e., adequate mix of obfuscation techniques such as dynamic code loading, bytecode encryption and reflexion) to what we call a *bootloader sequence* to be able to hide it (Figure 4.2, ❹).

Bootloader sequence variants. To hide the *bootloader sequence*, KillerDroid stores the *bootloader* in an external file that is either a `.DEX` file compiled to be concealed within the *host* application (named thereafter *external java bootloader*, See Figure 4.3, ❶`A`), or a `.so` file, i.e. a native (C/C++) dynamic library (named thereafter *external native bootloader*, Figure 4.3, ❷`C`). Finally, as we do not know beforehand the effect of externalizing the bootloader in a file, KillerDroid also generates a *bootloader* directly included into the bootstrap sequence, and named consequently *internal java bootloader* (See Fig-

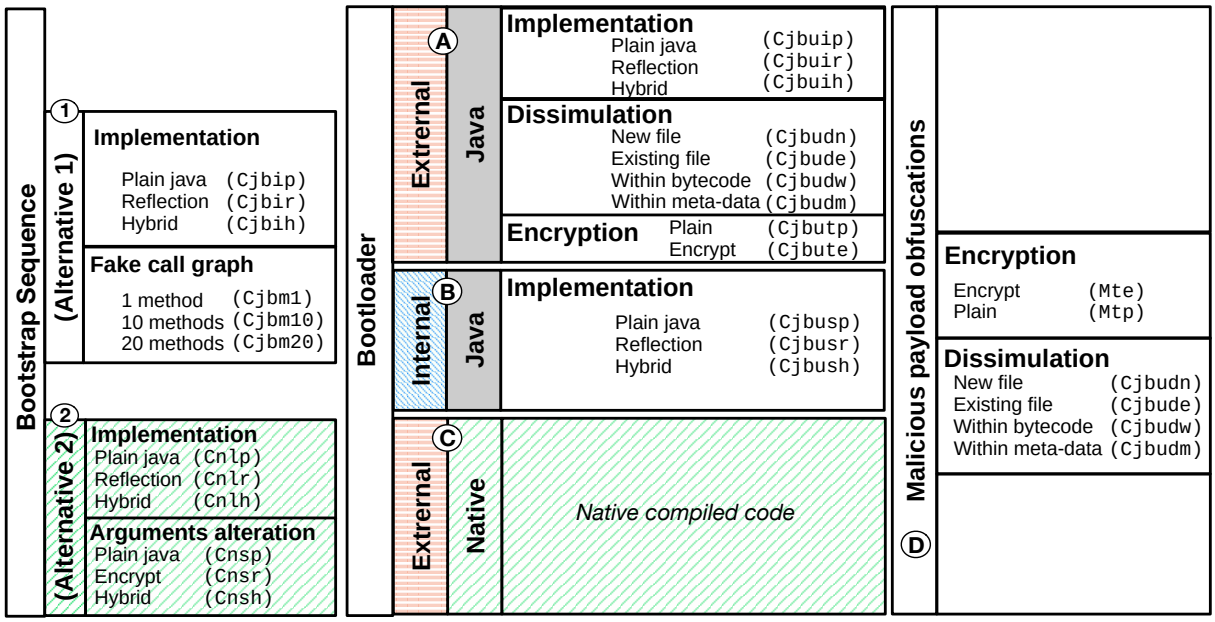

Figure 4.3 – Bootstrap and bootloader sequences

ure 4.3, ❶B).

External bootloaders, depending on whether they are Java or native based, require a different bootstrap sequence as specific methods must be used for loading either bytecode (with the `DEXClassLoader` method) or native code (with either the `System.load` or the `System.loadLibrary` methods). However, these methods are thoroughly searched by scanners as they are well known to be sensitive and used by malware [190]. In particular, based on static analysis techniques, scanners try to collect arguments passed in the aforementioned methods to determine the path of the file to be loaded to retrieve and analyze it afterwards. To bypass this kind of security checks, if an external bootloader is in Java, KillerDroid will additionally: (i) encrypt and/or dissimulate the bootloader payload into the host application, and (ii) generate the corresponding bootstrap sequence to deobfuscate it at runtime. In case of a native bootloader, KillerDroid encrypts method arguments, that are decrypted at runtime via the bootstrap sequence, avoiding any static analyzers to further extract/scan the native code.

Once the bootloader is loaded, a custom class loader is setup to load first and foremost classes from the malicious bytecode (Figure 4.2, ❺). To avoid malicious bytecode to be extracted/scanned, it is in turn encrypted or not, and dissimulated into the host application in a way similar as external java bootloaders (Figure 4.3, D).

Dissimulation. To avoid any suspicion, KillerDroid needs to dissimulate bytecode

files from java external bootloaders, and/or malware payloads. Accordingly, KillerDroid may perform first an AES encryption, to then choosing one among the four following possible methods to achieve dissimulation (Figure 4.3, `D`), i.e. injecting bytecode payload into: (i) a *new file* with a random name, (ii) in an *existing file* from the *host*'s assets directory, (iii) *metadata* of *host* assets' file images capable of holding metadata (e.g. `.png` files), (iv) application *bytecode* from the *host*. In the last two methods (i.e. *metadata* and *bytecode*), to maximize the chances to be as much as possible unsuspicious, the bytecode payload, intended to be hidden, is translated into a `BASE64` string split into a random number of chunks with a certain length threshold to avoid getting suspicious repetitive long chunks.

Accordingly, KillerDroid generates the corresponding bootloader to gather the spread chunks (Figure 4.3 `A,B,C`), and reconstruct at runtime the hidden bytecode previously split (Figure 4.2, ❻). For instance, when the *bytecode* method is chosen, the bootloader tracks the abstract syntax tree from the host to extract chunks, which have been previously dissimulated by the KillerDroid compiler into the bytecode of the host.

Merging subsidiary files

To run correctly, an Android application needs to execute its Dalvik bytecode along with subsidiary files such as the *AndroidManifest.xml* file, compiled and uncompiled resource files, Android assets, and native libraries (.so files). Hence, to produce a valid working AAE, its related application package must contain all files required by the *guest* malicious bytecode to be executed correctly, as well as the ones required by the *host* to start successfully. An additional challenge is to merge subsidiary files from both the *guest* and the *host* without being suspicious as scanners extract huge amount of metadata from these subsidiary files. Hence, a well known malware can easily be detected without even analyzing the application's bytecode with solely its metadata.

Manifest file role. The `AndroidManifest.xml` file is a cornerstone element of an Android application. Android applications are built upon basic units called *components* (e.g. Activity, Service, BroadcastReceiver, or ContentProvider), which are declared to the underlying Android system via the *manifest* file. Accordingly, the system and the declared components may interact altogether through a publish/subscribe mechanism with the use of *intent* messages. Each component is a potential entry point to the application, and components undeclared in the *manifest* are: (i) discarded from the application, (ii) unable to interact with anything. Further, any attempts to interact with an undeclared

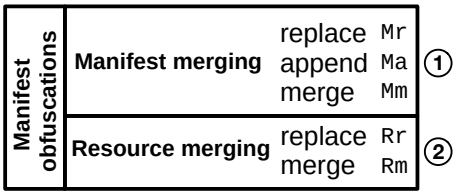

Figure 4.4 – subsidiary files obfuscations

component will result to an exception at runtime.

Challenges about updating a manifest. Irrespectively of how efficient KillerDroid is to dissimulate the bytecode payload of either the bootloader and/or the malware, the *manifest* must be updated accordingly to make the malware operational. On the one hand, dissimulating a *guest* malware within a *host* application can not be done without altering its *manifest* file. On the other hand, scanners commonly consider the *manifest* as a reference to collect application entry points, construct its related application call graph, and take a preliminary decision about the maliciousness of the application [21, 25, 197]. Updating the *manifest* (e.g., adding new components or removing existing ones) thus has an immediate impact on the way the application call graph is constructed and can increase suspicion. Consequently, the challenge is how to declare the *manifest* components of the dissimulated malware without increasing the chances of being detected as malicious.

Manifest obfuscation operations. KillerDroid supports three obfuscation operations to update *manifest* files: (i) *replace*, (ii) *append*, and finally *merge* (Figure 4.4 ❶). The first operation simply replaces the *host* application's *manifest* with the one from the *guest* malware, whereas the second operation consists of adding every component from the *guest* malware's manifest into the *host* ones. However, inside the manifest, declared components that come initially from the *guest* malware will not have a direct mapping with the accessible bytecode (as the payload bytecode from the malware has been hidden). Hence, if scanners check for consistency, they may suspect a malicious behavior. Finally, the third operation (*merge*) tricks the Android system by disguising malicious components from the *guest* as if they were components from the *host* application. KillerDroid collects all the components' names sorted by component type in both *host* and *guest manifest* files. For each malicious component found, KillerDroid tries to find a component of the same type in the *host* manifest and maps them together. If there is not enough benign components of a given type to be mapped with malicious components, random classes and names are generated from the *host* package name. In other terms, KillerDroid renames all malicious components' classes with the ones from the *host*. KillerDroid

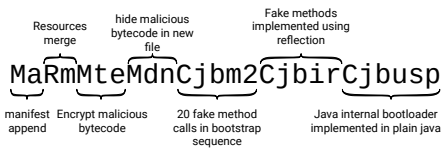

Figure 4.5 – A valid *tactic*

maintains a corresponding mapping table injected into the bootloader to be used by its underlying custom class loader to load adequately classes from the malware with their real names. Finally, KillerDroid always merges *guest* and *host* permissions within the *host* manifest file.

Merging uncompiled resource files. In a way similar to *manifest*, KillerDroid supports two obfuscations operations to deal with APK resources: (i) *replace*, and (ii) *merge* (Figure 4.4 ❷). The first one consists of replacing directly the *host* resources with those from the *guest* malware. The second one merges the resource directory from both APKs into a single one. If two resources have the same name, KillerDroid keeps the resource from the *guest* and discards the one from the *host*. Resources from the *host* are only used to fool scanners as bytecode from the host will be short circuited.

Obfuscation tactics

We now explain the generative approach on which we rely to generate massively AAEs for augmenting experimental datasets.

Obfuscation operations and Tactics. An *obfuscation operation* is a specific alteration to the *host* application during the instrumentation phase of the KillerDroid compiler. From Figures 4.3 and 4.4, it appears that there are 35 different obfuscation operations spread over 12 different groups. All obfuscation operations can independently be combined together to form *tactics* as long as they do not belong to the same group. As a result, a *tactic* corresponds to the list of obfuscation operations that KillerDroid must perform to silently intertwine a malware within a benign application. For instance, Figure 4.5 is an example of a valid *tactic* to be given as input to the KillerDroid toolchain. Further, note that due to the modular approach of KillerDroid, new obfuscation implementations can be easily added to enable KillerDroid to generate even more AAE variants.

Adversarial example variability. Given a couple of a malware and a benign application, we explore the space of all possible combinations of obfuscation operation, and

the number of AAEs it is possible to generate. Given the number of groups of possible obfuscation operations, their respective cardinality, and constraints between bootstrap sequences and bootloader (e.g. a native bootloader requires a dedicated bootstrap sequence) 12 096 possible tactics can be expressed to generate 12 096 unique variants.

4.2.2 Experimental Setup

This section explains the methodology used to evaluate KillerDroid against state-of-the-art Android malware detection scanners: Drebin [21], MaMaDroid [19], VirusTotal [42], and our own aggregator named KillerTotal to evaluate commercial scanners available directly from the PlayStore.

Dataset spawning with KillerDroid

Studied scanners are considered as black boxes, and hence KillerDroid has no prior knowledge on how scanners work. As a result, we exhaustively generate all possible combinations of obfuscation techniques for a couple of one benign *host* and one malware *guest* to generate malware samples.

Ground truth dataset. To asses a baseline for each studied scanner, we constitute *a ground truth dataset* (D_0), such as $D_0 = D_0^b \cup D_0^m$, made of two sub-datasets: (i) one composed of benign applications from the Google Play Store (D_0^b), and another composed of malware applications from several sources (D_0^m). Unlike several studies [37–39], Androzoo [158] is not used to collect benign applications as it relies on VirusTotal [42] to measure the maliciousness of an application, which is not acceptable as we compare KillerDroid to VirusTotal in our case studies. In contrast, we assume that most-downloaded apps in the Google Play official store, can be trusted as benign applications. Consequently, to build the D_0^b dataset, we consider the top 200 most downloaded apps in each category among the 33 categories (e.g., *games*, *photography*, ...) of the store. As Google tends to prevent automatic downloading of its applications by banning authenticated accounts on its platform, we reached only 5 269 applications over 6 600 expected before being banned. While this number of applications is enough for our case study, this limitation could be bypassed using techniques described in [158] to download more applications. We therefore consider a benign dataset D_0^b that contains 5 269 benign applications from the Google Play Store. Concerning the malware dataset D_0^m, it is composed of 5 953 malicious applications discovered between 2011 and 2019 gathered from: (i) Drebin

name		Guest application	Host application	# Samples	Error rate %	Total
D_0	D_0^b	*	*	5269	-	11222
	D_0^m	*	*	5953	0	
D_1	D_1^b	9574A64AD4	F11F862F69	12096	-	24072
	D_1^m	FE666E209E	F11F862F69	11856	1.98	
D_2	D_2^b	B2BF1CB046	42B0D3052A	12096	-	23709
	D_2^m	FE666E209E	42B0D3052A	11941	1.28	
$D_{2\prime}$	$D_{2\prime}^b$	401218235B	F11F862F69	12096	-	23929
	$D_{2\prime}^m$	F809CB4317	F11F862F69	12019	0.63	
D_3	D_3^b	638D8BFA47	4C4ACD530A	12096	-	23959
	D_3^m	F7659E8AB9	4C4ACD530A	11983	0.93	

Table 4.2 – Overview of datasets generated with KillerDroid

dataset [21] (samples from 2011 to 2013), (ii) Contagio dataset [88] (samples from 2011 to 2019), and (iii) a dataset of 200 manually certified ransomware from Androzoo [158] (samples from 2014 to 2015).

Adversarial examples datasets. We pick randomly one malware as a *guest* from D_0^m, and one benign application as a *host* from D_0^b to generate via KillerDroid a new dataset D_1^m composed of 12 096 variants of the selected malware *guest*.
Afterwards, to measure the capacity of machine learning scanners to detect malware variants after being retrained with already generated ones, we generate three new datasets D_2^m, $D_{2\prime}^m$ and D_3^m. The D_2^m dataset is generated using a different *host* application whereas, alternatively, $D_{2\prime}^m$ is generated using a different malware *guest* but with a same *host* compared to D_1^m. Finally, D_3^m is generated using both a new *host* and *guest*. Our purpose is to measure the impact on performance of ML based models over these new datasets, i.e. if we change either the *host* application only (D_2^m), the *guest* malware only ($D_{2\prime}^m$), and both of them (D_3^m). At its worst, KillerDroid generates less than 2% of non-functional variants (see chapter 5). Correspondingly, we end up with datasets composed of at least 11 856 and at most 12 019 samples as depicted in Table 4.2 and explained thereafter.

Avoiding sampling bias. *Sampling bias* arises when the distribution of training data does not reflect the actual environment that the ML model will be running on. If a machine learning model is trained with a subset of a population that is not representative of the whole population, the model can learn on features that characterize well a population subset but poorly the entire population. Therefore, a model trained with a *sampling bias*

will either wrongly predict unseen samples or well predict them but for wrong reasons. Concerning our datasets, *sampling bias* can occur if we do not take into account that benign applications also implement obfuscation techniques used in KillerDroid. Indeed, obfuscation operations used in KillerDroid are used by both: (i) malware to hide their malicious payload from scanners, but also (ii) benign applications when authors want to protect their intellectual property. Accordingly, to avoid *sampling bias*, KillerDroid generates datasets, named thereafter *sibling benign datasets*, by embedding existing benign applications into others, i.e., by replacing the malware *guest* by a benign application chosen randomly in D_0^b. Correspondingly, we have 4 sibling benign datasets D_1^b, D_2^b, $D_{2\prime}^b$ and D_3^b to balance their malicious counterparts. Hence, their size are aligned on the ones of their malicious counterparts to ensure well balanced training datasets, which is a strong requirement for training binary classifiers [198, 199]. Finally, we end up with 5 datasets $D_i, i \in [1;4] \wedge 2'$ containing up to 24 072 applications each, such as: $\left\{D_i = D_i^b \cup D_i^m\right\}$. The table 4.2 reports an overview of each generated dataset. To further avoid to introduce any false positives, *sibling benign datasets* are first evaluated against models trained with both datasets that include only non sibling benign applications, and with datasets that include also sibling ones to make scanners more robust.

Tools and malware detection scanners

Drebin. Drebin [21] uses static analysis to extract basic features from applications (e.g. *permissions*, *protected API calls* ...). Features are embedded in a feature vector using a binary encoding to represent the presence or absence of a given feature. A linear Support Vector Machine (SVM) is then applied to classify applications as either malicious or benign. In the literature [33, 39], Drebin is still considered as one of the most efficient machine learning based approach. As Drebin's authors did not release their source code, we did our own implementation of their approach according to their paper. More precisely, we use the Soot framework [167] to extract features, and use the same *one-hot-encoding* approach to encode boolean features into a feature vector. However, we improved their approach by extending it to two other algorithms: Random Forest and KNN.

MaMaDroid. Contrary to Drebin, MaMaDroid [19] is a ML scanner based on behavioral features similar to other past studies [116, 200]. Instead of extracting basic features, MaMaDroid extracts the entire application call graph using static analysis, and encodes the latter in an abstract representation by keeping only method package names. Afterwards, MaMaDroid builds Markov chains by calculating the probability for a method

to be called by another. Finally, calculated probabilities are used to build the application feature vector, to be used to train 3 different classification models with Random Forest, KNN and SVM.

VirusTotal. VirusTotal is an online platform, now owned by Google [201], which scans each file uploaded into the platform using a set of 60 commercial anti-viruses software. VirusTotal provides a final scores that correspond to the number of antivirus that flagged the file as malicious over the total number of antivirus. VirusTotal gives us the opportunity to test each AAE generated from KillerDroid against commercial scanners.

Performance measurements

A *true positive* (TP) is defined as a correctly classified malicious application, whereas, a *true negative* (TN) is defined as a correctly classified benign application. A *false positive* (FP) is defined as an incorrectly classified benign application, whereas a *false negative* (FN) is defined as an incorrectly classified malicious application.

Correspondingly, to evaluate scanners' performances, we use standard machine learning measures: *Accuracy, F1-Score* and therefore *Precision* and *Recall*. Accuracy is defined such as :

$$Accuracy = \frac{|TP| + |TN|}{|TP| + |TN| + |FP| + |FN|}$$

where $|TP|$, $|TN|$, $|FP|$, $|FP|$ are the number of predicted samples in each category. The accuracy corresponds to the number of correctly classified samples over the number of samples to predict (the total population). As we use balanced datasets, the accuracy score can be further assessed by the F1-Score:

$$F_1 = 2 * \frac{precision * recall}{precision + recall}$$

where $precision = |TP|/(|TP| + |FP|)$ and $recall = |TP|/(|TP| + |FN|)$. For a machine learning model to be accurate and useful, there is a trade-off in the metrics to maximize. Hence, on one hand, a model with a good recall, and bad precision will classify many applications as malware, even benign ones. On the other hand, a model with a good precision score and bad recall will classify many malicious applications as benign thus involving significant security issues. The F1-Score is a function of precision and recall that seeks an optimal score for both precision and recall. In case of balanced datasets

(same number of samples in each class), F-Score allows to assess the accuracy score's quality for a binary classifier. When performing model training, we try to maximize both accuracy and F1-Score. We use a K-fold cross validation with K=10 to reduce the threat of an overfitting situation. Another issue to solve is to avoid *overfitted* models. *Overfitting* happens when the trained model does not generalize well from the training data to new unseen data. One way to overcome *overfitting* effect is to perform K-cross fold validation, i.e. randomly dividing the training dataset in K equal size random subsets, and train the model K times using $K-1$ subsets for training, and 1 subset for testing. In our evaluation, we perform a 10-cross fold validation for each model training, which is a good compromise to spot potential overfitting problems and avoid too small subsets. Also, for each training run, we shuffle together all samples from input datasets, and select 60% of samples for training and 40% for testing the trained model. Finally, for each training of a model, we provide the assessment scores obtained by the best classifier found during the cross-validation process. Then, we explain scores obtained when asking trained models to predict new unseen data.

4.2.3 Evaluation

Our evaluation focuses on the following research questions:

— **RQ1 - Evasion**. Can scanners detect AAEs generated with KillerDroid?

— **RQ2 - Adversarial retraining**. Can scanners learn to detect AAEs generated with KillerDroid?

— **RQ3 - Vulnerabilities**. What are the most efficient obfuscation operations to defeat scanners ?

RQ1 - Evasion

MaMaDroid, Drebin and VirusTotal are evaluated against the ground truth dataset D_0.

Performance of scanners on the initial malware dataset. MaMaDroid and Drebin are each evaluated with their Random Forest, SVM and KNN (with $k = 1$ and $k = 3$) learners using the best set of *hyper-parameters* found during the grid search process. W.r.t. VirusTotal, we have submitted each sample from D_0, and have counted the number of detected samples.

Scanner	Accuracy	F1-Score	Precision	Recall
Drebin				
KNN (k=1)	99.38	99.45	98.91	100.00
KNN (k=3)	99.22	99.32	98.64	100.00
RF	99.01	99.14	98.42	99.86
SVM	98.99	99.11	98.33	99.91
MaMaDroid				
KNN (k=1)	75.92	77.42	78.98	75.92
KNN (k=3)	78.88	79.96	82.57	77.52
RF	82.22	85.49	76.87	96.29
SVM	83.60	85.79	81.10	91.06
VirusTotal				
Detection	97.22	97.45	95.03	99.98
KillerTotal				
Detection	99.66	99.67	99.35	100.0

Table 4.3 – Accuracy, F1-Score, precision and recall obtained by MaMaDroid and Drebin on the D_0 dataset (11 222 samples).

1. *Drebin:* As shown in Table 4.3, all Drebin models trained with D_0 obtain an Accuracy and a F1-Score above 99% which is 5% more accurate than the model initially trained in the original paper. This improvement comes from our re-implementation of Drebin. It collects a larger set of features than the original Drebin (e.g., *sources* and *sinks* [202], and more *protected API calls* thanks to Axplorer [45]).

Hyper-parameter search for Drebin and MaMaDroid

To get the best baseline, we improved published results obtained by the authors of both Drebin and MaMaDroid by performing an *hyper-parameter optimization* via a grid search approach for each studied algorithm. Table 3.5 gives the obtained hyper-parameter values to get the best trained model. Note that our experiments are done with *Scikit-learn* [154], so hyper-parameter names correspond to the *Scikit-learn* API.

	Parameters	Mamadroid	Drebin
Random Forest	n_estimators	2000	1500
	max_depth	40	30
	min_samples_split	10	4
	min_samples_leaf	10	10
	max_features	sqrt	auto
SVM	C	500	100
	kernel	linear	rbf
	gamma	auto	auto
KNN	n_neighbors	3	3
	weights	uniform	distance
	leaf_size	20	20
	p	2	2

Table 4.4 – Best hyper-parameter combinations found for each algorithm with both approaches: Drebin and MaMaDroid

Accordingly, in Table 4.3, models trained with a KNN algorithms obtain a F1-Score of 75.92% for 1-NN and 78.88% for 3-NN, which is 20% less than results obtained originally. The model trained with the Random Forest algorithm obtains a F1-Score of 85.49% which is 10% less than the original paper. Notice that although the authors have disqualified SVM from their experience, we show that this algorithm performs better than the others when the right combination of hyper-parameters is used for training. Therefore, our model trained with SVM obtains an accuracy of 83.60% with a F1-Score of 85.79% which is 4.23% more precise than the model trained with the Random Forest algorithm, but still 11% less compared to the best results obtained from the MaMaDroid's original dataset. Our *hyper-parameter optimization* process allows to improve by up to 2% the accuracy primarily reached with models trained with default hyper-parameter values.

2. *MaMaDroid:* MaMaDroid's models yield F1 scores that are 10 to 19 percentage points less than those reported in the original paper. This is explained by the fact that we use a different dataset which include more recent malware and applications (2019). MaMaDroid is indeed not robust to the evolution of the training dataset over time, and hence suffers from *temporal bias* [35, 39, 160].

3. *VirusTotal:* As the ground truth dataset D_0 is composed of known malware, which have already all been submitted and reported as such to VirusTotal, the platform correctly detected almost every submitted samples. It reaches an accuracy of 97.22%, and a F1-Score of 97.45% (see details in table 4.3). Among the 5 952 malware submitted to VirusTotal in the D_0 dataset, one malware was not detected at all and 5950 of them were detected by more than five anti-viruses.

4. *KillerTotal:* All malware is detected. However, only 4207 malware were detected by more than five anti-viruses. Note that KillerTotal has only 7 anti-viruses compared to 60 for VirusTotal. However, all malware has been detected by at least one anti-virus.

Evaluating scanners robustness against KillerDroid. All three scanners are trained with D_0. They are asked then to predict the class, i.e. benign or malicious, of all samples from dataset D_1. Results are given in Table 4.5. Note that the feature extraction processes of Drebin and MaMaDroid sometimes fail, in particular due to bugs in the Soot Framework on which both approaches rely. These issues prevent to build the feature vector of certain samples, which is why the number of samples tested can sometimes be

Scanner	Accuracy	F1-Score	Precision	Recall	FN	FP	TN	TP
Drebin over D_1								
KNN (k=1)	47.95	0.00	0.00	0.00	**10620**	0	9785	0
KNN (k=3)	47.95	0.00	0.00	0.00	**10620**	0	9785	0
RF	47.95	0.00	0.00	0.00	**10620**	0	9785	0
SVM	47.95	0.00	0.00	0.00	**10620**	0	9785	0
MaMaDroid over D_1								
KNN (k=1)	48.92	54.08	50.82	57.80	**4482**	5941	3844	6138
KNN (k=3)	49.12	54.71	50.96	59.06	**4348**	6035	3750	6272
RF	51.79	62.69	52.49	77.82	**2355**	7482	2303	8265
SVM	49.22	61.13	50.81	76.72	**2472**	7889	1896	8148
MaMaDroid over D_3								
KNN (k=1)	49.59	6.59	60.60	3.48	**10056**	236	9762	363
KNN (k=3)	50.31	55.21	50.12	60.01	**4167**	5978	4020	6252
RF	50.71	62.32	51.09	79.87	**2097**	7966	2032	8322
SVM	50.35	55.18	51.16	59.90	**4178**	5959	4039	6241
VirusTotal over D_1								
Threshold=1	67.53	61.05	76.43	50.82	**5921**	1887	10126	6120
Threshold=2	53.77	9.40	84.30	9.40	**10908**	211	11802	1133
Threshold=5	49.94	0	0	0	**12041**	0	12013	0
KillerTotal over D_1								
Threshold=1	70.24	57.75	99.66	40.65	7179	17	12066	4917
Threshold=2	50.14	0.66	100.00	0.33	12056	0	12083	40
Threshold=5	49.97	0.00	0.00	0.00	12096	0	12083	0

Table 4.5 – Trained with D_0, tested with either D_1 and D_3

lower than the theoretical size of the dataset. For dataset D_1, 15.65% of feature vectors could not be generated for either DREBIN or MAMADROID. However, as it is irrelevant for VIRUSTOTAL, all samples has been successfully uploaded to evaluate VIRUSTOTAL.

1. *Drebin:* All trained models from the DREBIN wrongly classify the totality of malicious samples from D_1 (10 620 `FN`), while being able to correctly classify all benign samples (9 785 `TN`). Models reach an accuracy of 47.95%, but fail to correctly predict all malware variants, leading to a precision of 0. Surprisingly, whatever the obfuscation tactics used, DREBIN is unable to detect any AAE. All features collected from the AAE are strongly similar to those extracted from the original host, and do not reflect the actual behavior of the AAE. Therefore, DREBIN is not able to build feature vectors that differentiate enough variants from the host application used to build AAEs.

   To better understand the outcome of DREBIN models, we perform a Principle Component Analysis (PCA) to obtain a two dimensional representation of feature vectors calculated by DREBIN (Figure 4.6). The objective is to estimate the proximity of vectors among the *host* application ($\vec{H_1}$), the *guest* malware ($\vec{G_1}$), and the ones corresponding to AAEs obtained by KILLERDROID ($\forall \vec{x} \in D_1^m$). It appears that KILLERDROID performs a good translation of $\vec{G_1}$ enabling to sys-

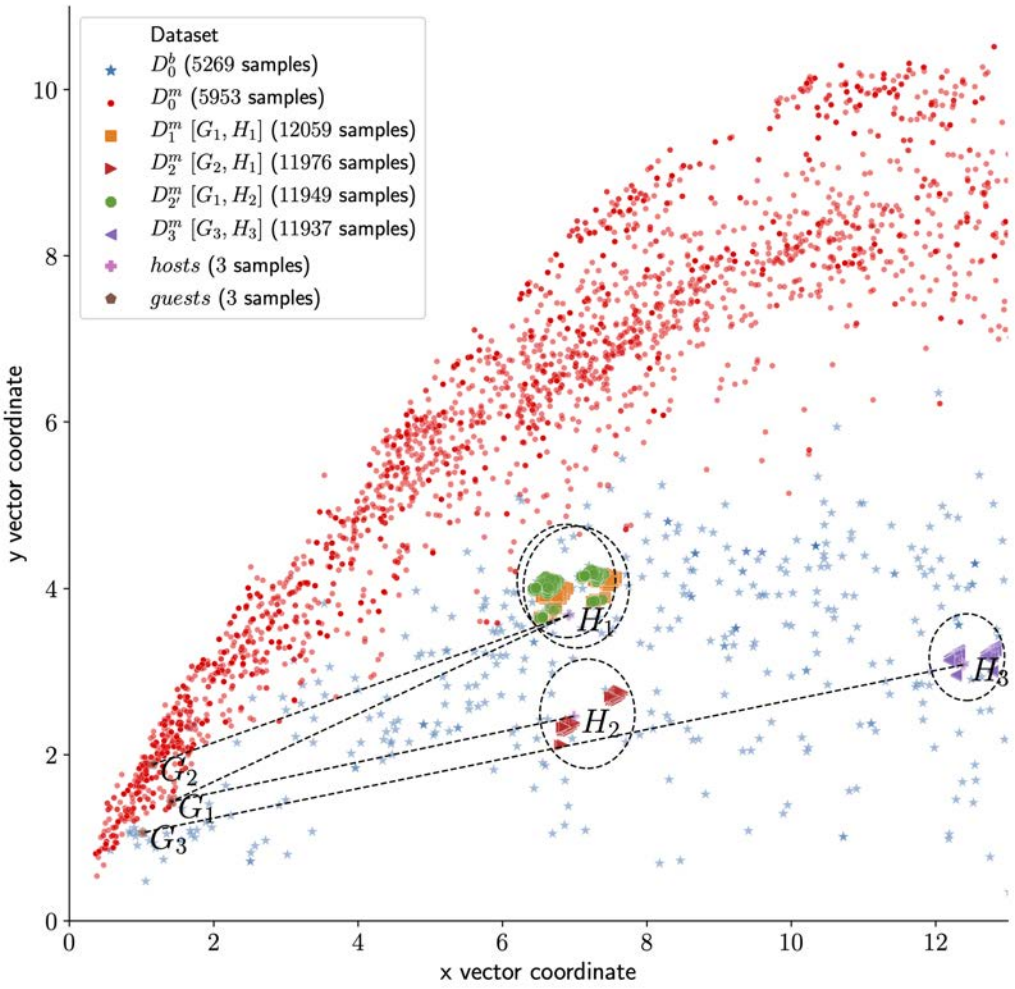

Figure 4.6 – Projections of feature vectors in a 2D space

tematically bypass Drebin. $\forall \vec{x} \in D_1^m$ is clustered around the *host* $\vec{H_1}$, that itself is located inside the D_0^b cluster clearly separated from the D_0^m cluster. It validates that feature vectors $\vec{x} \in D_1^m$ (AAEs) are both too close to the *host* $\vec{H_1}$, and too far from all feature vectors $\vec{y} \in D_0^m$ (malware used to train Drebin) to be classified as malicious. To further confirm these results, we stress tested Drebin with 3 other datasets D_2^m, $D_{2'}^m$ and D_3^m. For each experiment, we get the same result, a cluster of AAEs centered around the *host* (either $\vec{H_1}, \vec{H_2}$, or $\vec{H_3}$) inside the D_0^b cluster, and far away from the malware cluster D_0^m used to train Drebin (whatever the guest used $\vec{G_1}, \vec{G_2}$, or $\vec{G_3}$), leading to malware misclassifications.

We further use a Jaccard similarity metric $J(\vec{A}, \vec{B})$ to get insights about the proximity between two feature vectors $\vec{A}$ and $\vec{B}$ as Drebin's feature vectors represent categorical values. Correspondingly, $J(\vec{G_1}, \vec{H_1}) = 0.058$, whereas the average of the Jaccard similarity:

$$\overline{J_{D_1^m}} = \frac{\sum_{i=1}^{i=|D_1^m|} J(\vec{x_i}, \vec{H_1})}{|D_1^m|} = 0.72$$

with a standard deviation of 0.040. These numbers corroborate that KillerDroid produces AAEs that Drebin cannot distinguish from the host application.

2. *MaMaDroid:* MaMaDroid performs much better than Drebin. At best, when tested over D_1^m, MaMaDroid gets a F1-Score around 77%. In other terms, it fails to detect 2 374 malware over 10 620 in the best case (i.e. 22.35% of malware). However, this result comes at a high cost, as in return, 7 482 benign applications have been considered as malware over 9 785 (i.e. $76,46\%$). It explains why the recall reaches at best 77.82%, but only with a precision score of 51.79%. Malware detection is improved (decrease number of `FNs`) at the expense of a misclassification of benign applications (increase number of `FPs`) and *vice versa*. To evaluate if the behavior of MaMaDroid remains constant in terms of results, we repeat the experiment with another dataset D_3 generated using a different *guest* and *host* (see table 4.5). We observe the same phenomenon as previously. The accuracy is not better, and minimizing the number of `FPs` drastically increases the number of `FNs`. The precision reaches at best 60.60% with 1-NN as there is only 236 `FPs`. However, the corresponding recall reaches 3.48% as there is correspondingly 10 056 `FNs`. Conversely, with the best F1-score at 62.32% with RF, there is only 2 097 `FNs` (i.e. 19.74% of malware undetected), but it comes with a huge spike of 7 966 `FPs` (i.e. 81.41% of benign applications misclassified) leading to a precision of only

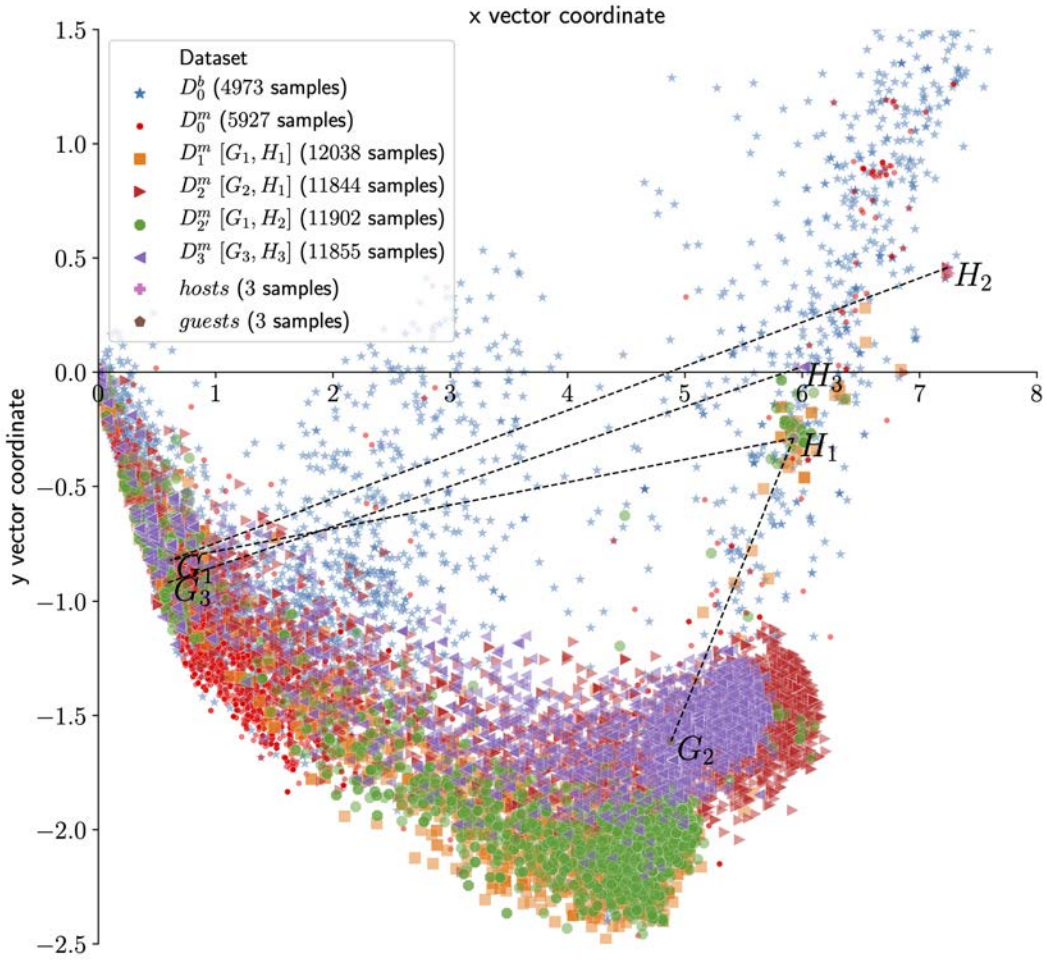

Figure 4.7 – Projections of feature vectors in a 2D space

51.09%.

In a way similar to Drebin, we perform a PCA (see figure 4.7) over feature vectors calculated by MaMaDroid to get a better understanding of the impact of obfuscation tactics. Interestingly, feature vectors $\vec{y} \in D_0^m$ and $\vec{b} \in D_0^b$ are not clustered but spread over the 2D space with an area of overlap, explaining the precision score that reaches only 81.10% at best with the ground truth dataset D_0. It leads one to think that MaMaDroid may have difficulties to have a good accuracy as feature vectors from D_0^m and D_0^b are not clustered but spread and mixed. This is confirmed with feature vectors $\vec{x} \in D_1^m$ from samples generated by KillerDroid. They are not clustered either around their *host* $\vec{H_1}$, but are spread over the 2D space. Further, they are spread and mixed with either feature vectors $\vec{y} \in D_0^m$ or $\vec{b} \in D_0^m$ used to train MaMaDroid, explaining the bad accuracy and precision that stay roughly around 50 in average. In this particular case, the Jaccard similarity metric is useless here as there is no observed clusters. Additionally, we stress tested MaMaDroid with 3 others datasets D_2^m, $D_{2'}^m$ and D_3^m. For each experiment, we observe the same trend whatever the *host* ($\vec{H_1}, \vec{H_2}$, or $\vec{H_3}$) or the *guest* ($\vec{G_1}, \vec{G_2}$, or $\vec{G_3}$) selected. The phenomenon of non-clustering and spreading highlights the importance of the impact of obfuscation operations over MaMaDroid. Some obfuscation tactics operate a translation of the *guest* that enables to bypass MaMaDroid, and other do not (See Section 4.2.3 that provides deeper analysis to determine the most efficient obfuscation operations). Whatever, with both an accuracy and a precision around 50% MaMaDroid is unusable.

3. *VirusTotal:* In practice, a high number of benign applications embed *adware*, which often lead benign applications to be misclassified as malware by anti-viruses from VirusTotal. Consequently, several studies [19, 203] consider that a sample is a malware only if exactly at least 5 anti-viruses over 60 flagged it as malicious. In contrast, in our studies, we give the results for either at least 1, 3 and 5 anti-viruses (i.e. with a threshold $th = 1$, $th = 3$ or $th = 5$). Hence, if we take a threshold $th = 5$ antivirus (as usually observed in the domain) to consider a sample as a malware, we observe that 100% of samples from D_1^m bypass the detection, leading to F1-score, precision and recall to 0% (Table 4.5). The accuracy reaches around 50% as the number of the `TN` is equal to the number of benign applications with no misclassifications (i.e. number of `FP` is equal to 0). At the opposite, if we are very conservative, and consider at best a threshold $th = 1$, F1-score reaches 61.05%

with a recall of 50.82%. Mostly, the number of `FN` raised down to 5 921 (i.e. mostly 50% of AAEs have been correctly classified as malware), however it comes at the price of 1 887 `FP`. VirusTotal gets a good precision score (76.43%), mainly due to the number of benign samples correctly classified, but still is unable to detect 50% of malware.

4. *KillerTotal:* In a way similar to VirusTotal, with a threshold $th = 5$, we observe that 100% of malware from D_1^m bypass the detection, leading to F1-score, precision and recall to 0%. The accuracy is also of around 50% as no benign applications has been misclassified. From a more conservative perspective (with $th = 1$), performance of KillerTotal are roughly similar to VirusTotal but with a slightly better precision as there are less `FP` compared to VirusTotal. Nevertheless, there are more malware misclassified; `FN` increased slightly to reach 7179 compared to 5921 for VirusTotal. To summary, the versions of our selected set of anti-viruses directly available in the Google PlayStore get roughly similar performances compared to VirusTotal, and are, as well as VirusTotal, vulnerable to AAEs generated by KillerDroid.

Our experiments revealed that KillerDroid can produce AAEs that evade detection from state of the art scanners in the literature and antivirus systems. 100% of KillerDroid-generated malware variants are not detected by Drebin although the guest and host in the AEEs were involved in the training data of the scanner. While MamaDroid fails to detect a relatively lower number (∼19%), it also misclassifies 81% of benign apps. Commercial antivirus systems hosted at VirusTotal fail to detect 50% of KillerDroid-generated AAEs.

RQ2: Adversarial retraining

Models from Drebin and MaMaDroid are now retrained with $D_0 \cup D_1$ (i.e., adding some AAEs to the initial dataset) to evaluate their robustness, i.e. their ability to detect new unseen AAEs generated by KillerDroid. In this scenario, VirusTotal and KillerTotal are discarded since we do not have access to their training process.

Estimating performance on ground-truth Results are given in Table 4.6. Both accuracy and F1-Scores for Drebin's models are above 99%. Drebin is able to correctly detect AAEs that share the same *host* and *guest* as the ones used to retrain the models. Re-

garding MaMaDroid, accuracy scores range from 56.64% to 62.10%, and F1-Scores range from 59.14% to 62.49% for all retrained models. Unlike Drebin, MaMaDroid struggles to correctly detect unseen AAEs even if they are generated from the same *host* and *guest* as the ones used to retrain the models. MaMaDroid trained over $D_0 \cup D_1$ leads to poorer performance compared to when it is only trained with D_0. Surprisingly, increasing the number of samples into the training dataset (up to 39 614) has a negative impact.

Scanner	Accuracy	F1-Score	Precision	Recall	Dataset size
Drebin					
KNN (k=1)	99.81	99.82	99.74	99.91	30092
KNN (k=3)	99.76	99.78	99.61	99.94	30092
Random Forest	99.58	99.61	99.34	99.89	30092
SVM	99.62	99.64	99.40	99.89	30092
MaMaDroid					
KNN (k=1)	59.08	59.69	60.63	58.78	34852
KNN (k=3)	59.80	59.14	62.12	56.43	34852
Random Forest	62.10	62.49	63.79	61.24	34852
SVM	56.64	60.47	57.04	64.35	34852

Table 4.6 – Performance scores on datasets $D_0 \cup D_1$.

Evaluating scanners robustness against KillerDroid. D_2 and $D_{2\prime}$ datasets, described in Section 4.2.2, are used to evaluate the robustness of retrained models. As all samples from D_2 and $D_{2\prime}$ share either the *host* or the *guest* with the samples from D_1, which has been used to retrained the models, we expect from Drebin and MaMaDroid a high detection rate. Prediction results are given in table 4.7.

1. *Drebin:* Drebin models predict all samples from D_2 and $D_{2\prime}$ as benign, leading to an accuracy of 48.51% with a F1-Score of 0 as all malicious samples have been misclassified. As a result, unfortunately, retraining the Drebin's models has no effect over the malware detection rate although the new AAEs share either the *host* or the *guest* with the samples included into the training dataset used. As previously, regardless of obfuscation tactics used by KillerDroid, Drebin has misclassified systematically AAEs from D_2^m and $D_{2\prime}^m$. Vector features calculated by Drebin do not enable to significantly differentiate malware and benign applications.
2. *MaMaDroid:* Retraining the MaMaDroid's models by increasing the number of samples into the training dataset, passing initially from 11 222 up to 34 852 samples, leads to counter productive results as performances are worst than previously. When tested with D_2, MaMaDroid gets, at best, a F1-Score around 57.70%. Pre-

Scanner	Accuracy	F1-Score	Precision	Recall	FN	FP	TN	TP
Drebin over D_2 (same malware as D_1)								
KNN (k=1)	48.51	0.00	0.00	0.00	10250	0	9657	0
KNN (k=3)	48.51	0.00	0.00	0.00	10250	0	9657	0
Random Forest	48.51	0.00	0.00	0.00	10250	0	9657	0
SVM	48.51	0.00	0.00	0.00	10250	0	9657	0
Drebin over $D_{2\prime}$ (same host as D_1)								
KNN (k=1)	48.97	0.00	0.00	0.00	10419	0	9998	0
KNN (k=3)	48.97	0.00	0.00	0.00	10419	0	9998	0
Random Forest	48.97	0.00	0.00	0.00	10419	0	9998	0
SVM	48.97	0.00	0.00	0.00	10419	0	9998	0
MaMaDroid over D_2 (same malware as D_1)								
KNN (k=1)	38.81	24.68	33.69	19.47	8254	3928	5729	1996
KNN (k=3)	39.12	24.45	33.86	19.13	8289	3830	5827	1961
Random Forest	51.89	57.70	52.71	63.73	3717	5861	3796	6533
SVM	39.57	30.83	37.54	26.15	7569	4461	5196	2681
MaMaDroid over $D_{2\prime}$ (same host as D_1)								
KNN (k=1)	50.26	55.08	51.08	59.77	4192	5964	4034	6227
KNN (k=3)	50.58	55.78	51.33	61.09	4054	6036	3962	6365
Random Forest	50.59	61.83	51.03	78.43	2247	7841	2157	8172
SVM	50.98	62.07	51.28	78.62	2228	7781	2217	8191

Table 4.7 – Performances of MaMaDroid, Drebin on dataset D_2 and $D_{2\prime}$ when trained with datasets $D_0 \cup D_1$ (39 614 samples).

cisely, it fails to detect 3 717 malware over 10 250 in the best case (i.e. 36.26% of malware), but with 60.69% of benign applications misclassified. It seems that the *host* randomly chosen from D_0^b acts as a better attack vector for the *guest*. When tested with $D_{2\prime}$, performances are better than with D_2: at best 2 228 malware over 10 419 are undetected (i.e. 21.38%), associated with a spike of $77,82\%$ of benign applications misclassified. Additionally, changing the malware *guest* does not seem to change anything in terms of performances. Overall, although the new AAEs share either the *host* or the *guest* with the samples included into the training dataset initially used (i.e. $D_0 \cup D_1$), it has no positive effects over the malware detection rate.

Proliferation of new generations of variants. It is well admitted that in the field of machine learning, it is usually preferable to retrain models with as many samples as possible to obtain better results [35, 39]. However, so far, in the latest experiments, MaMaDroid obtains opposite results. To investigate this trend, we retrain all models

for both Drebin and MaMaDroid with a training dataset made of $D_0 \cup D_1 \cup D_2 \cup D_{2'}$ accounting for a ground truth dataset of 82 932 samples (i.e., versus 39 614 for $D_0 \cup D_1$, and 11 222 for D_0). Results are given in Table 4.8. As previously, concerning the ground truth, Drebin performed as well as previously observed. However, results from MaMaDroid are drastically worst. For instance, from a F1-score of 85.79% with an associated recall of 91.06% over D_0 (i.e. 11 222 samples), it has been raised down to a F1-score of 54.46% with an associated recall of 57.90% over $D_0 \cup D_1 \cup D_2 \cup D_{2'}$ (i.e. 82 932 samples), highlighting the negative impact of the retraining for MaMaDroid' models.

Scanner	Accuracy	F1-Score	Precision	Recall
Drebin				
KNN (k=1)	99.67	99.68	99.56	99.81
KNN (k=3)	99.67	99.68	99.53	99.84
Random Forest	99.67	99.68	99.45	99.92
SVM	99.38	99.40	99.76	99.04
MaMaDroid				
KNN (k=1)	55.92	63.46	54.82	75.33
KNN (k=3)	56.08	47.36	60.58	38.88
Random Forest	56.31	54.46	57.90	51.40
SVM	52.60	54.00	53.28	54.74

Table 4.8 – Performance yielded by models trained with datasets $D_0 \cup D_1 \cup D_2 \cup D_{2'}$ (82 932 samples).

The dataset D_3 is used to test the retrained models. Table 4.9 confirmed previous observed results. Even after being retrained with a high number of variants coming from several different couple of *guest* and *host*, machine learning models are unable to correctly classify new unseen AAEs generated from a different couple. These results demonstrate the incapacity of either Drebin or MaMaDroid models to catch up with the massive generation of new variants by KillerDroid. The more MaMaDroid is trained with new samples, the worse the results are.

Even retrained with some AAEs, state of the art machine learning based scanners show their limitations when assessed against KillerDroid-generated datasets. For all retraining scenarios, Drebin systematically failed to detect 100% of the generated variants. Concerning MaMaDroid, the performance decreases gradually as the number of samples in the training dataset increases. These results suggest that these scanners are not as effective as portrayed in the literature.

RQ3: Weaknesses

Finding best obfuscation operations using simulated annealing. As we consider studied scanners as black boxes, we use KillerDroid to brute force the solution space

Scanner	Accuracy	F1-Score	Precision	Recall	FN	FP	TN	TP
Drebin over D_3								
KNN (k=1)	50.00	0.00	0.00	0	10604	0	10605	0
KNN (k=3)	50.00	0.00	0.00	0	10604	0	10605	0
Random Forest	50.00	0.00	0.00	0	10604	0	10605	0
SVM	50.00	0.00	0.00	0	10604	0	10605	0
MaMaDroid over D_3								
KNN (k=1)	48.25	50.74	48.41	53.30	4952	6023	4582	5652
KNN (k=3)	48.55	47.93	48.52	47.35	5583	5328	5277	5021
Random Forest	50.02	59.34	50.01	72.94	2869	7731	2874	7735
SVM	48.53	60.11	49.07	77.56	2379	8537	2068	8225

Table 4.9 – Performances of MaMaDroid and Drebin on dataset D_3 (23 959 samples) when trained with datasets $D_0 \cup D_1 \cup D_2 \cup D_{2\prime}$ (trained with 82 932 samples).

by generating AAEs for all possible combinations of obfuscation operations. Accordingly, we have no clues about which obfuscation tactics can favor the detection of an AAE by a given scanner. While some combinations of obfuscation operations can facilitate evasion, other combinations can have the opposite effect and produce easily detectable malware.

Simulated annealing. A simulated annealing algorithm [204] allows to approximate the *global optimum* of a given function when the search space is discrete. In our case, we seek

Algorithm 1 Simulated annealing

$s \leftarrow s0$
for k = 0 through k_{max} **do**
  $t \leftarrow temperature(k, k_{max})$
  $s_{new} \leftarrow mutate_state(s)$
  $e \leftarrow get_energy(s)$
  $e_{new} \leftarrow get_energy(s_{new})$
  if $P(e, e_{new}, t) \geq random(0, 1)$ **then** $s \leftarrow s_{new}$

the combination of obfuscation operations that produce the highest rate of undetected malware over the total number of malware that have this combination for a given scanner. We define $O = \{o_0, o_1, \ldots, o_i\}, i \in [0; 34]$ as the set of all obfuscation operations. We define s such as $s \subset O$ a state that represents one combination of obfuscation operations and $s_0 = \varnothing$ the initial state as an empty set. Then, we define k as a step, and k_{max} the total number of steps to be taken to succeed. We define the function `mutate_state(s)` that randomly adds or removes an obfuscation operation in state s and returns a new state

s_{new}. We define a `temperature(k)` function such as

$$temperature(x) = \alpha * \frac{k_{max}}{k+1}, \alpha \in R$$

where α is a factor that defines the speed at which the algorithm must converge toward a solution. We define a function `get_energy(s)` that calculates the *error rate* as the following:

$$get_energy(s) = \frac{\text{\# of undetected malware compliant with s}}{\text{total number of malware}}$$

Finally, we define the transition probability function $P(e, e_{new}, T)$ such as:

$$P(e, e_{new}, T) = \begin{cases} 1, & \text{if } e_{new} \leq e \\ e^{-\frac{(e_{new}-e)}{T}}, & \text{otherwise} \end{cases}$$

which rules the probability of moving from one state to another. At each step, the algorithm chooses a new solution s_{new} which is close to the current state s. Then, depending on the calculated error rate e_{new} for s_{new}, the algorithm decides to move to the new solution or stay with the current one (s) based on the probability function P. The *temperature* in the algorithm represents a slow decrease in the probability of choosing worse solutions during solution space exploration. Combined with a high enough value of k_{max}, this property allows to search for the global optimal solution.

Finding most effective obfuscation operations. We run the simulated annealing algorithm for both VirusTotal and MaMaDroid (trained with D_0) on three sub-datasets $D_1^m \cup D_2^m \cup D_{2'}^m$ that account for 29 170 AAEs. We omit Drebin as no malware samples were actually detected as malicious by the scanner.

For VirusTotal, we observe that AAEs solely based on *dynamic code loading*, i.e. without additional obfuscation techniques, such as *manifest merging* (Mm) or *malicious payload encryption* (Mte), are easier to detect. As such VirusTotal correctly detected 16.58% of AAEs for D_1^m (resp. 16.35% for D_2^m, and 15.17% for $D_{2'}^m$) without encryption (Mtp), and by replacing the host manifest (Mr). Inversely, when AAEs are encrypted with either a manifest merge or append (Mm, Ma), only 1.22%, 0.77% and 0.57% (resp. for D_1^m, D_2^m, $D_{2'}^m$) samples are correctly detected.

According to the algorithm, we find that using obfuscation operations from the set $\{Ma, Mm, Mte\}$, corresponding to *manifest append*, *manifest merge* and *malicious pay-*

load encryption, gives the lowest error rate for VirusTotal and account for 7928 undetected malware over 9725 malware generated with these types of modifications. This result emphasis the fact that VirusTotal anti-viruses are vulnerable to *bytecode encryption* and *AndroidManifest.xml* merging manifests with *append* or *merge* techniques. In other words, chances of creating a variant undetectable by VirusTotal are maximized if it has been generated with *tactics* that include couples $\{Ma, Mte\}$ or $\{Mm, Mte\}$ (as Mm and Ma are exclusive). Among AAEs generated using this association of obfuscations operations, 81.5% of them have been undetected by VirusTotal and account for 27% of tested samples.

Concerning MaMaDroid, we find that operations on the manifest files are the most meaningful obfuscations operations to generate undetectable AAEs. More precisely, results of the simulated annealing algorithm show that $\{Ma, Mm\}$ are the most effective operations and 6 417 (13% of tested samples) over 8 718 (29% of tested samples) adversarial examples generated with one of these OOs are not detected. These results confirm our intuition: when the host manifest keeps host components, the MaMaDroid algorithm use these entry points to build the call graph with the host application dead code. Therefore, the resulting feature vector is similar to the benign host application. Among AAEs generated using either Ma or Mm, 46.00% of them have been undetected by MaMaDroid and account for 13% of tested samples.

The simulated annealing algorithm emphasis the fact that three obfuscation operations (Mm, Ma, Mte) are paramount to generate an undetectable AAE. However, weights of these OOs over detection rates of MaMaDroid and VirusTotal hide the impact of other OOs that can also be important to bypass detection. Therefore, we also investigate the correlation between detection results and the presence of an obfuscation operation in tactics used to generate AAEs in $D_1^m \cup D_2^m \cup D_{2'}^m$ for both MaMaDroid and VirusTotal. The figure 4.8 shows the correlation between obfuscations operations operated by KillerDroid and the prediction of MaMaDroid and VirusTotal on AAEs. The darker the box (smaller coefficient correlation), the more the obfuscation operation helps the variant to be undetected. Conversely, the clearer the box (higher coefficient correlation) the more the obfuscation operation makes the variant easy to detect. Firstly, we observe that OOs found by the simulated annealing algorithm are highly correlated to undetection. Correlation coefficients (cc) for Mm and Ma are below -0.33 for VirusTotal and -0.38 for MaMaDroid and coefficient for Mte is equals to -0.31 for VirusTotal. Secondly, we observe that the place where the malicious payload is hidden (Md, Md) has also an impact

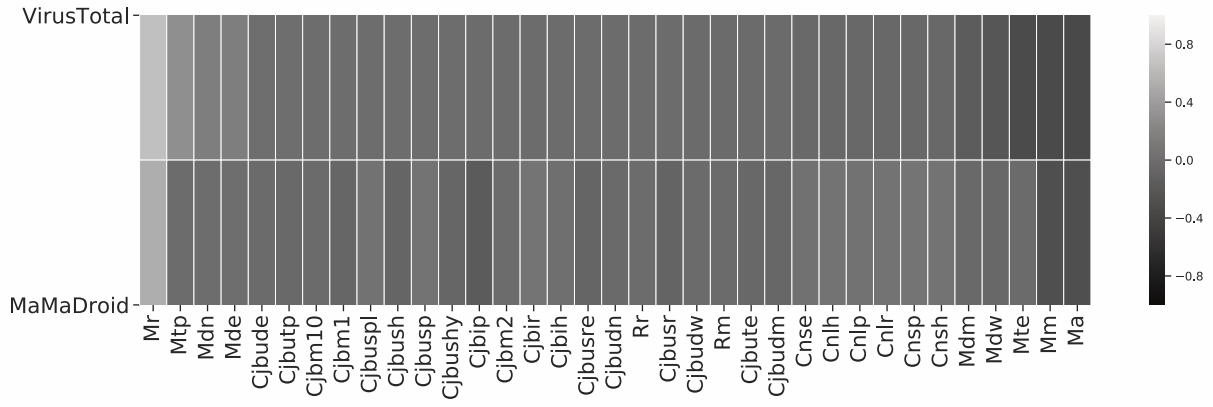

Figure 4.8 – Correlation between detection prediction and obfuscation operations for MaMaDroid and VirusTotal on dataset D_1^m

on the VirusTotal detection rate. Thus, hiding the byte code either in pictures meta-data Mdm (cc -0.14) or within the host application byte code Mdw (cc -0.20) is more correlated to undetection than simply hiding the payload in a random file (Mdn - cc 0.17, Mde - cc 0.18). Regarding MaMaDroid, we observe that most of obfuscation operation do not have a strong correlation with detection results except for *manifest* related obfuscations.

Effectiveness of bootloader strategies. We show that the bootloader strategy used to generate adversarial examples also has a significant impact on scanners detection rates. Again, we discard Drebin because the approach did not detect any adversarial examples from KillerDroid. As presented in figure 4.9, AAEs generated with the *native* bootloader are 30% less detected than AAEs with a Java bootloader by the MaMaDroid approach. This can be explained by the fact that the loading of native code is less verbose and requires very few modifications to the host application bytecode. Moreover, dynamic libraries (*.so* files) are not taken into account by MaMaDroid. Furthermore, we observe that AAEs generated with a *native* bootloader are 10% less detected than AAEs with a Java bootloader by VirusTotal. We make the hypotheses that anti-viruses embedded in VirusTotal as well as MaMaDroid do not take into account native code loading and do not statically analyze code in dynamic library files.

MaMaDroid and VirusTotal are especially weak in face of obfuscations that target the manifest file. VirusTotal antiviruses are very vulnerable to the encryption of the dynamically loaded bytecode. Hiding the malicious payload in either the host byte code or in picture meta-data also plays a role towards creating undetectable variants for VirusTotal. Finally, AAEs generated with a native bootloader are 30% less detected by MaMaDroid

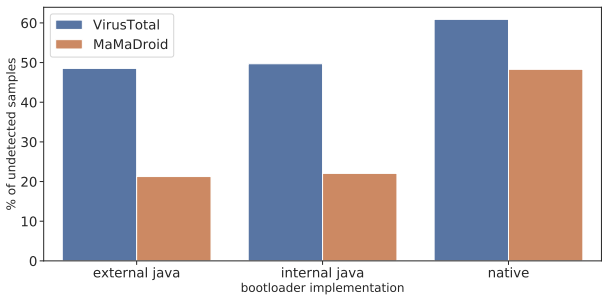

Figure 4.9 – Percentage of undetected AAEs depending on the bootloader strategy used.

and 10% by VirusTotal compared to AAEs generated with a Java bootloader.

4.3 Conclusion

In this chapter, we have demonstrated that KillerDroid can effectively generate Android adversarial examples at scale. With example datasets created with KillerDroid we have further proven that the state-of-the-art in Android malware detection, both from research and industry, are particularly vulnerable to such attacks. Similar toolchains or adversarial approaches may have already landed in wrong hands, and one can only imagine the disastrous effect it could have on the Android ecosystem security. We designed KillerDroid to avoid such situation. KillerDroid allow researchers to generate massive challenging datasets to confront current malware detection scanners to the latest obfuscation techniques observed in the wild. We believe that such initiative will greatly improve performances of existing scanners and will help further to detect 0-day malware.

However, to evaluate KillerDroid we make two optimistic assumptions that could introduce a bias in our experiment. Firstly, we do not take into account the validity of crafted adversarial examples. While we effectively put in evidence vulnerabilities of existing scanners, it is of little interest if crafted malware variants are not functional. In the coming chapter, to overcome this unrealistic hypothesis, we discuss the benefits of testing the validity of generated malware variant. We further propose an approach to systematically asses the validity of malware variant crafted from samples of several malware families.

Secondly, to evaluate the resistance of commercial antivirus to KillerDroid generated variants, we make use of the VirusTotal platform. While this platform offers a scalable api to test a large number of sample rapidly, it is known to host lightweight versions of an-

tivirus products with limited capabilities. As such, evaluating antivirus products through VirusTotal may give an unrealistic snapshot of their performances. In the next chapter, we first propose a discussion regarding the limits of cloud-based antivirus solutions. Then, we present a new original approach to evaluate fully-fledged antivirus products directly on Android devices.

Chapter 5

Increasing the quality of ground truth datasets on Android

An important step towards implementing and evaluating new malware detection approaches is the creation of ground truths for benign and malicious applications. The efficiency of a malware detection method is directly correlated [35, 36] with the quality of the dataset used to train and test it. Nonetheless, at least two factors can influence the quality of a ground truth dataset:

— **Inaccurate labels.** Inaccurate labels, i.e. malware samples mislabeled as benign or vice versa, in the training dataset of a classifier can decrease the accuracy of the resulting model [36].

— **Unrealistic samples**. Unrealistic samples, such as non-working or outdated applications, can artificially increase the accuracy of a model at test time but the model may suffer from overfitting and be unable to detect new unforeseen malware [35, 39].

One accurate manner to cope with such issues is to manually analyse and label application samples. However, the number of samples required to efficiently train a machine learning-based detection approach makes manual analysis and labeling unfeasible. Such situation emphasizes the need for (i) more accurate methods to automatically label application samples and (ii) an efficient manner to automatically vet application samples to ensure their quality. In this chapter, we present two novel approaches aim to increase the quality of ground truth datasets on Android by addressing the two aforementioned claims. We first present KillerTotal, a novel approach to evaluate individually the publicly available mobile versions of the seven most efficient scanners from VirusTotal. Secondly, we present KillerScreenshot, an original technique to automatically certify that synthetically crafted samples produced by adversarial production toolchains are actually working.

5.1 Limits of antivirus aggregators solutions to build ground truth datasets

To fight against new malware, considerable efforts are deployed to bring realistic and efficient malware detection both on online platforms [18, 42, 140] or directly on end devices [21, 81, 205]. Nevertheless, attackers are still able to exploit the vulnerabilities of both signature-based and ML-based detection methods. Signature-based scanners remain vulnerable to 0-day malware and ML-based scanners are targeted by adversarial attacks [33, 107]. Signature and ML-based scanners share a common weakness: their efficiency depend on the quality and the size of ground truth datasets used to differentiate legitimate and malicious applications. Traditional signature-based scanners must constantly update their knowledge database with signatures of the latest malware found in the wild in order to keep an acceptable detection rate. Besides, ML-based scanners must ensure that datasets used for model training contain a sufficient number of correctly labeled samples in realistic proportion to avoid suffering from *spatial* and *temporal bias* [35]. Moreover, to cope with adversarial attacks [33, 107], ML-based approaches must constantly retrain their models with new data to take into account samples that evade detection.

However, automatically collecting new unforeseen malware is an arduous task and requires human intelligence to manually label new samples found in the wild. To circumvent this problem and bring freshness to their datasets, several companies such as VirusTotal (owned by Google) [42] leverage crowdsourcing by maintaining collective submission services that allow individuals to freely upload files. In exchange for the collected data, these platforms offer the user to quickly diagnose uploaded files with their scanners. For example, for each file uploaded to VirusTotal, the platform executes more than 60 antivirus engines from third-party vendors as well as several static and dynamic analysis tool to capture insightful information from the studied content. Such content is stored as metadata along with the file and publicly available. By default, VirusTotal does not explicitly label a sample as benign or malicious. Given a file, VirusTotal returns the prediction (i.e. labels) of the 60 antivirus products, and it is up to platform users to interpret these results in order label their samples. As such VirusTotal, records around 1 million distinct new files submissions every day, mainly Windows and Android executables [206].

This kind of solution is profitable for all antivirus vendors as each of them can receive a copy of every uploaded file, thus guaranteeing an inexhaustible source of data [207].

More importantly, VirusTotal helps to rise the global IT security level by sharing its file corpus with premium cybersecurity customers and research teams worldwide. For example, in the research domain, a vast majority of studies proposing an ML-based malawre detection approach [21, 22, 37–41] rely on VirusTotal to automatically label samples and build ground truth datasets. However, recent studies [23, 43] raised several VirusTotal limitations suggesting that such labelling method may introduce a bias in ground truth datasets built upon it.

Such limitations can be induced by the fact that VirusTotal actually runs restrained versions of commercial antivirus engine on its platform. Version of scan engines and tools provided by third party companies to VirusTotal differ from versions used in production by security companies. As stated by VirusTotal itself [42], scan engines proposed by the platform are not fully fledged and correspond to engine versions restrained on purpose by third-party companies for commercial and/or technical reasons. The fully-fledged products versions of these engines, desktop and mobile programs, are more likely to use behavioural analysis as well as monitoring system events to get more insights about the analyzed samples.

Consequently, relying on VirusTotal to label samples and build ground truth datasets may be not as precise as relying on fully-fledged products sold by commercial security companies. We believe that building ground truth datasets by labelling samples with mobile versions of antivirus engines from commercial companies could greatly benefit to malware detecition methods based on machine learning. In this work we pave the way for a more accurate approach to (i) evaluate the efficiency of commercial antivirus and (ii) build more reliable malware datasets. Firstly we show, with a dataset of more than 10000 manually vetted Android application samples, that antivirus products hosted on VirusTotal are less efficient than their fully-fledged mobile application counterpart. In particular we show that several mobile version of the antivirus are 10 times more accurate than the VirusTotal hosted version. Then, we demonstrate that even if fully-fledged versions of antivirus are not able to defeat adversarial attacks done with KillerDroid, they are slightly more resistant (see section 4.2).

5.2 KillerTotal: Vetting grand public Antivirus products

In this work, we present a novel approach to evaluate individually the publicly available versions of the seven most efficient scanners from VirusTotal, as the ones aggregated on VirusTotal are usually constrained and/or lightweight versions compared to their mobile application counterparts on the Google Play Store. The 7 antivirus have been selected following a multi-criteria basis from the most to the least important: (i) providing the best performance on the VirusTotal platform, (ii) having no dependencies on arm native libraries (as we use x86 cpu), (iii) being the most downloaded on the Google Play Store. Hence, we are evaluating the following 7 anti-viruses: *AegisLab*, *Bitdefender*, *DrWeb*, *Panda Security*, *Zoner AntiVirus*, *G Data*, and *Malwarebytes*. Our system, which we call KillerTotal, is a framework based on the emulation of the Android operating system. KillerTotal automates the installation and execution of known antivirus mobile application products on running emulators. Afterwards, KillerTotal evaluates the antivirus in terms of precision and accuracy by sequentially installing apk samples on the underlying emulator. KillerTotal provides a large-scale deployment platform that can be used to evaluate the performances of antivirus products by testing thousands of applications in an acceptable amount of time.

5.2.1 Mobile Antivirus products

Mobile antivirus products are Android applications released and distributed by cybersecurity companies through the Google Play Store. Companies sell these applications as scanners that can detect a new threat on a user's device in real time. As such, antivirus products are provided with multiple features such as file system scans or the scan of every third party application installed on the device. Antivirus products on the market can be thought as a black box as we have no insight on how the application works to perform malware detection. However, most of antivirus products have similar approaches to detect that a new application has been installed on the system. Antivirus applications subscribe to the `ACTION_PACKAGE_ADDED` system intent which allow them to be woken up by the system when a new application is installed on the device. Products use the permission `READ_INTERNAL_STORAGE` to have access to the newly installed application file and perform their detection algorithm. For safety reasons as well as a better user experience, antivirus

products are systematically informing the user when they finished the scan of a newly installed application by telling him if the application is either safe to use or dangerous. KillerTotal leverages on this behaviour shared by all antivirus products to evaluate them at scale.

5.2.2 Approach

KillerTotal can be thought as a framework which makes it possible to generalize as much as possible the evaluation of a large number of antivirus. Antivirus products are developed and maintained by independent companies, therefore each antivirus application has its own behaviour. This prevents to have a generic approach to evaluate all antivirus products. Instead, KillerTotal is organised around a global API that abstracts all common steps, as well as several drivers, one per antivirus product. The global API is responsible for executing tasks such as fetching APKs, installing them on the emulator, cleaning the emulator environment, communicating with the database, collecting antivirus responses, etc. Antivirus drivers are responsible for performing specific tasks inherent to each antivirus such as clicking on user interface buttons, cleaning the antivirus cache, monitoring antivirus notifications, etc.

KillerTotal is built to be resilient to all external events that can occur on a running Android operating system. Indeed, many perturbations, such as device notifications, lost of connectivity, unpredictable application behaviours can prevent or alter the evaluation of an antivirus or put the emulator in a dead lock state. To prevent such problems, KillerTotal is designed around a lifecycle with multiple checks to ensure that the emulator is in the best possible condition to monitor the antivirus response when an apk is installed. An overview of the KillerTotal lifecycle is presented on figure 5.1.

Initialisation phase

The main aim of the initialisation phase is to prepare the environment to evaluate the antivirus product in best possible conditions. KillerTotal works in concert with a virtual machine running the Android operating system. When executed, KillerTotal begins by starting an Android Emulator and waits for it to be up and ready. The bootstrap time of the emulator depends on the resources allocated and can be up to 30 seconds. Then, it checks that the emulator has a working internet connection, which is required as several antivirus products are using remote servers to perform application analysis.

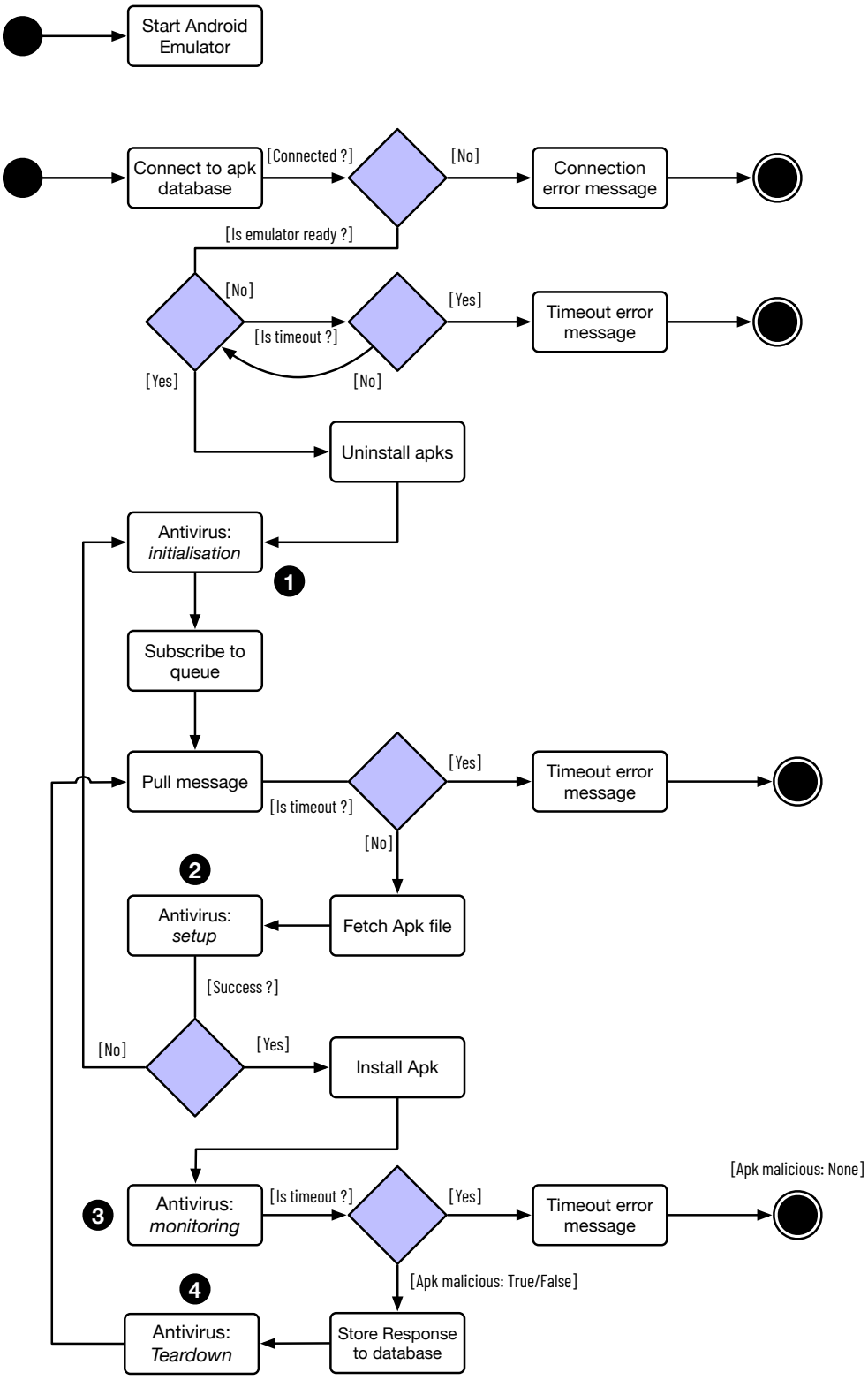

Figure 5.1 – KillerTotal lifecycle overview

Afterwards, KillerTotal connects itself to the database in which apks are stored. Once the Android Emulator is up, KillerTotal cleans the environment and removes all third party applications that might be installed on the system such as Google play services. Once the emulator is ready, the side-car program performs a series of checks to ensure that the device is in an acceptable state to run an experiment on it. Next, KillerTotal installs the antivirus product to evaluate and applies the initialisation step specific to the product (see 5.1, step ❶). The initialisation step is responsible for performing all actions required to put the antivirus product in a suitable state to be correctly monitored when it analyses a newly installed application. This step is crucial as no antivirus product can work out of the box by simply installing the application on a device. On the Android operating system, a permission is only granted to an app at execution time and not a installation time. As such, no antivirus product can detect newly installed applications without being executed and initialised at least once. Moreover, antivirus applications may initially ask the user to sign in or to activate a trial licence before running correctly. Finally, some antivirus applications products will not work before having performed a scan of the entire device filesystem. Once the product is initialised, KillerTotal enters in a loop state, acting as a consumer in a queue management system. KillerTotal pulls messages from this queue that contains the necessary information to fetch apks files to be tested by the antivirus. For each received message, KillerTotal performs steps that are specific to each antivirus product.

Antivirus setup

The setup phase (see 5.1, step ❷) is required to put the antivirus product ready to analyse a new application application sample. Some antivirus are constantly performing background tasks that alter its cache and local storage so KillerTotal takes care of cleaning such artifacts before going any further. Moreover, KillerTotal ensures that the antivirus application is running in the foreground to ensure that it is not idle or disabled by the Android system. Once done, KillerTotal installs the application sample to be analysed by the antivirus.

Antivirus monitoring

The monitoring phase (see 5.1, step ❸) consists of observing the Android Emulator until the antivirus application produces an event telling the user that the sample application has been analysed. Such event can have many different forms such as system

notifications or Android toasts. KillerTotal observes the emulator's screen periodically (the period depends on the antivirus behaviour) until it detects a new event related to the antivirus product. When such event occurs, KillerTotal parses the content of the produced event and stores the response to the database before triggering the teardown phase.

Antivirus teardown

The teardown phase (see 5.1, step ❹) consists of both producing an output for the tested application sample and cleaning the test environment to make it ready for the next cycle. When antivirus products scan a newly installed application, they write metadata into their local database and file storage. They also produce user interface content within the app itself that can interfere with next scans. KillerDroid takes care of cleaning the environment as much as possible to avoid altering next cycles with erroneous data.

5.2.3 Evaluation

Implementation details

KillerTotal uses Nomad, an open-source workload orchestrator developed by HashiCorp, which provides a flexible environment to deploy our samples on top of an abstracted infrastructure. More precisely our Nomad instance federates a cluster of 6 Intel(R) Xeon(R) Gold 6136 CPU nodes that accounts for 600 GB of RAM and 124 cores at 3.0 Ghz. Each antivirus is installed into a Pixel 2 Emulator with Android 8.1 running itself into a Docker container. Besides the emulator, the Docker container contains a side-car python process responsible for initialising and running the lifecycle of the antivirus product. The side-car program leverages on the *Android debug bridge* (`adb`) to communicate with the emulator. Each emulator has a daemon (`adbd`) running as a background process which execute commands it receives on the device. Adb is a *client-server* program that allows a *client* to send commands to a *server* that manages communication between the client and the daemon. Adb allows to run various commands on the device such as installing applications or get device information. When an adb client is started, the client first check if an adb server is running on the machine and starts one otherwise. On startup, the server binds to local port 5037 and listen for incoming commands from the client. Afterwards, the server also scans a range of ports on the machine to setup connection with emulators' `adbd` daemon advertised on these ports. When the docker image is instantiated as a container, the en-

trypoint starts the Android emulator in background and the side-car program thereafter. Then, the side-car program immediately starts an adb server and client and waits for the emulator to be up and running. Each antivirus product as its own side-car implementation, levering its own *monkeyrunner* script. *Monkeyrunner* is a python library that allows to control the user interface and events of an Android Emulator through `adb`.

Antivirus	Initialisation time	APK test time
DrWeb	23.48	5.81
AegisLab	44.91	5.47
BitDefender	22.53	3.16
GData	52.49	18.71
Zoner	19.11	23.37
Malwarebytes	140.59	14.45
Pandas Security	24.83	4.05

Table 5.1 – Average process time for each KillerTotal supported antivirus

For scalability reasons, Android emulators running in each KillerTotal container is not restarted after each apk tested. Indeed, the time required to boot the emulator and initialise the antivirus is incompressible, thus each container do it only once at startup. An x86 Android Emulator started with 2GB of RAM and 3 cores at 3.0Ghz will take 90 seconds to start. Besides, the average time required to initialise each antivirus is reported in table 5.1. As such, restarting the emulator for each tested apk could take up to 4 minutes, which makes it unrealistic to test thousands of apks in an acceptable amount of time. Instead, a KillerTotal container boots the Android emulator once, initialises the antivirus and then sequentially test apks by cleaning the emulator environment between each test. The table 5.1 reports time duration statistics for each antivirus supported by KillerTotal. The time required to test one apk depends on the antivirus: *BitDefender*, the fastest one, requires 3.16 seconds to test one apk on average. *Zoner*, the slowest one, will take 23.37 seconds on average to test one apk.

KillerTotal leverages on its scalable architecture (see figure 5.2) to compensate for the time taken by antivirus to analyse an application. There is one docker image per antivirus product supported by KillerTotal that can be deployed in as many containers as necessary. KillerTotal uses a publish-subscribe messaging pattern as a message queue system to advertise to containers apks that must be analysed. To begin with, the user runs a publisher program (see figure 5.2, step ❶) that is responsible for pushing messages to topics for all apks to be analysed. A topic is created for each antivirus product

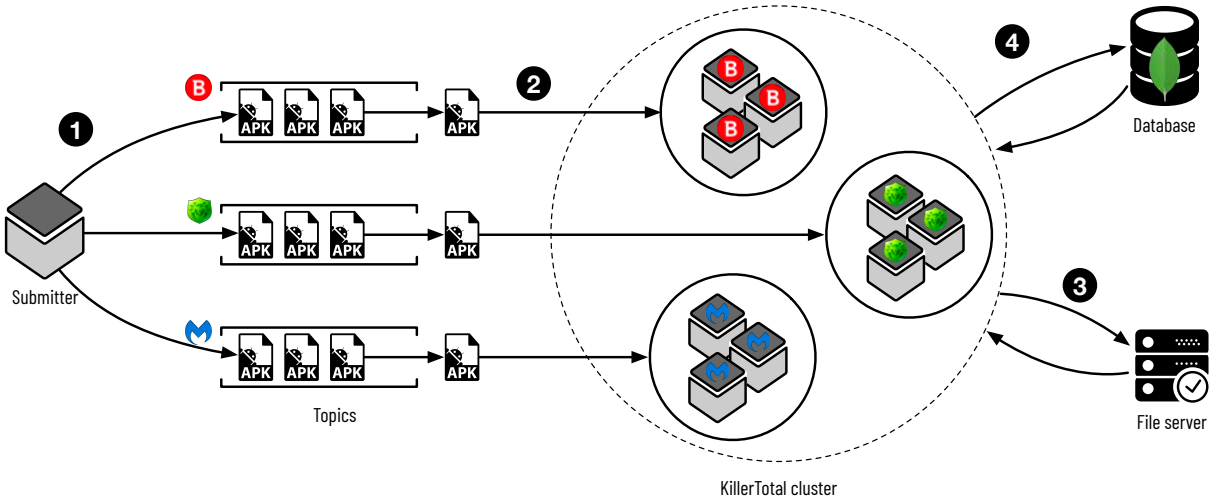

Figure 5.2 – KillerTotal infrastructure overview

supported by KillerTotal and user can choose topics to which messages should be published. When started, KillerTotal containers will act as subscribers by subscribing themselves to the topic that concerns them. As such containers responsible for evaluating the *BitDefender* antivirus will pull messages from the related *BitDefender* topic. When a container is ready to analyse a new apk, it pulls a message from its topic (see figure 5.2, step ❷), then fetches the apk on the file server (see figure 5.2, step ❸) and stores results to the mongo-db database when finished (see figure 5.2, step ❹). The pub-sub messaging mechanism is setup in a *shared* mode which ensure that messages will be delivered in a round robin distribution across subscribers. For a given topic, any given message is delivered to only one subscriber. If a subscriber fails to correctly process the message (the apk is not correctly analysed by the antivirus), the message is not acknowledged and will be redelivered later on. Such architecture allows KillerTotal to scale infinitely horizontally by adding as much containers as possible within limits of available resources. Each KillerTotal container is allocated with 2 cores and 4GB of RAM. As such, the cluster can run 62 containers in parallel. Given these performances, KillerTotal can test 5.9 applications per second and takes 32 minutes to evaluate a dataset composed of 11222 samples.

VirusTotal vs. KillerTotal

In this experiment, our goal is to compare the effectiveness of the seven antivirus supported by KillerTotal against their implementation counterpart on VirusTotal.

To do so, we build a dataset of 11222 applications composed of 5269 benign and 5953 malware samples. As we can not rely on neither VirusTotal nor KillerTotal to assert the maliciousness of a sample, all applications in the dataset are manually vetted. Benign applications are collected by downloading the top 200 applications of each categories on the Google Play Store. Malware applications are collected from well known datasets used by the research community: the contagio dataset [88], the Drebin dataset [21] and 200 ransomwares manually vetted from Androzoo [158].

	Accuracy		**Precision**		**Recall**		**F1-Score**	
Approach / **Antivirus**	KillerTotal	VirusTotal	KillerTotal	VirusTotal	KillerTotal	VirusTotal	KillerTotal	VirusTotal
AegisLab	**99.96**	97.23	99.93	99.77	100.00	94.99	99.96	97.32
BitDefender	**99.10**	49.50	99.61	99.66	98.71	4.87	99.16	9.29
DrWeb	**99.80**	98.29	99.78	99.62	99.89	97.16	99.84	98.37
GData	**97.15**	55.11	99.86	99.89	94.04	15.46	96.86	26.77
Malwarebytes	**98.50**	46.92	99.78	0.00	97.80	0.00	98.78	0.00
Panda	**99.33**	49.02	99.98	100.00	98.85	3.97	99.41	7.63
Zoner	55.53	**63.20**	100.00	99.67	17.88	30.78	30.34	47.03

Table 5.2 – Detection scores obtained by KillerTotal and VirusTotal on a dataset of 11222 samples

As a first step, a script is responsible for uploading all samples from the dataset to VirusTotal and collect the results for the seven antivirus selected. Then each sample is sent to the KillerTotal cluster to be analysed by the seven mobile antivirus executed on Android emulators. To obtain the detection scores of antivirus for both approaches, we calculate the number of wrongly classified samples among our dataset of 11222 samples. The table 5.2 reports results obtained for both platforms. Results show that over seven antivirus evaluated, six of them obtain a better accuracy with KillerTotal than with their VirusTotal counterparts. *AegisLab* and *DrWeb* obtain rather equivalent results for both platforms: *AegisLab* reports an accuracy of 99.96% and 97.23% on VirusTotal, *DrWeb* reports an accuracy of 99.80% on KillerTotal and 99.29% on VirusTotal. However, four over seven antivirus obtain far better results on KillerTotal than VirusTotal. For example, the *BitDefender* mobile application reports an accuracy of 99.10% while its VirusTotal counterparts reports an accuracy of only 49.50%, suggesting that half of submitted samples are misclassified. Indeed, *BitDefender* obtains a recall score of 4.87% on VirusTotal which indicates that almost all malicious samples have been undetected by the antivirus. On the contrary, the recall score of *BitDefender* on KillerTotal is 98.71%, indicating that almost all malicious samples has been detected by the mobile application. The antivirus *Zoner* is the only one to obtain better results on VirusTotal

compared to its mobile application on KillerTotal with accuracy scores of 63.20% and 55.53% respectively. However, these scores are much lower than other antivirus tested, suggesting that the *Zoner* antivirus (both on mobile or in the cloud) is rather unreliable.

Performances of KillerTotal on KillerDroid adversarial examples

Antivirus	Accuracy	Precision	Recall	F1-Score	TN	TP	FN	FP
AegisLab	49.42	0.0	0.00	0.00	11819	0	12096	0
BitDefender	34.99	100.0	3.58	6.91	5843	433	11663	0
DrWeb	50.00	0.0	0.00	0.00	12095	0	12096	0
GData	49.52	0.0	0.00	0.00	11868	0	12096	0
Malwarebytes	66.16	100.0	33.12	49.76	11811	4006	8090	0
Panda	49.95	0.0	0.00	0.00	12067	0	12093	0
Zoner	49.89	0.0	0.00	0.00	12040	0	12094	0

Table 5.3 – Detection scores obtained by KillerTotal on a dataset of adversarial examples generated by KillerDroid

As a second experiment, we further want to evaluate the capacity of KillerTotal to detect adversarial examples generated by KillerDroid (see section 4.2). To do so, we use KillerDroid to generate a new dataset of 24192 samples: 12096 samples generated by weaving a malicious payload into a benign application and 12096 samples generated by weaving a benign payload into a benign application. Notice that antivirus sometimes fail to analyse an application, as such, our final dataset is composed of 23915 samples instead of 24192. The results obtained are reported in table 5.3. For five over seven antivirus, we observe an accuracy around 50%. This can be explained by the fact that these antivirus correctly classify all benign samples but fail to detect all malicious samples. We observe that *Malwarebytes* obtains slightly better results with an accuracy of 66.17% as it is able to correctly detect 4006 malicious samples over 12096. To summarise, mobile versions of our selected set of antivirus get roughly similar performances to VirusTotal (see section 4.2.3), and are, as well as VirusTotal, vulnerable to adversarial examples generated by KillerDroid.

5.3 KillerScreenshot: Improving the quality of malware datasets

One limitation of many works [33, 97, 98] studying the detection of adversarial examples on Android is that they do not verify that adversarial examples they create are actually functional. Instrumenting and altering malware bytecode at large scale most often leads to nonfunctional samples. There is a negative tradeoff between altering malware bytecode, and bypassing antivirus/anti-malware scanners [104, 106]. For instance, with MRV [106], at best, from a dataset of 409 malware, authors are able to generate only 696 working variants. We believe that it is of limited interest to generate challenging datasets for antivirus systems composed of mostly nonfunctional samples. In this context, we augmented our work on KillerDroid with an approach that allows to systematically assert the validity of each generated adversarial example. This approach allowed us to demonstrate that among all adversarial examples generated by KillerDroid so far (48 934 samples), 98% of them are fully working.

5.3.1 Approach

In this work, we propose an original technique to automatically certify that samples produced by adversarial production toolchains are actually working. To achieve this, we designed a toolchain, KillerScreenshot, which dynamically verifies that the behaviour of a crafted adversarial example is consistent with the behaviour of the original malware used to create it. Android malware in the wild are very diverse and account for thousands of different behaviour that can sometimes be hard to observe. For this reason, we limit our approach to visible malware behaviours, that said behaviours that have an impact on the device user interface, such as done by Ransomware, Scareware or banking trojans. Such malware families are known to display an overlay or an activity to block the user interface or encourage the user to enter its banking credentials. KillerScreenshot leverages on the structural similarity index measure (SSIM) [208] to compare the screen activity of adversarial examples with the original malware. More precisely, the SSIM allows to compare the *structure* between two images instead of only performing a pixel-to-pixel difference. This is required in our approach because several small perturbation can appear on screen between two screeshot such as the clock ticking, or contextual text displayed. KillerTotal provides a large-scale deployment platform based on Android Emulators that allow

to deal with the number of AEs that KillerDroid can generate rapidly.

To certify the validity of an AE with respect to the original malware, KillerScreenshot takes to inputs (i) the apk of the original malware and (ii) the apk of the target AE. Then, KillerScreenshot communicates with the Android Emulator to install and install applications provided as input. More precisely, KillerScreenshot applies a sequence of two phases (i) context initialisation and (ii) AE verification.

Context initialisation.

The main aim of the context initialisation phase is to obtain the screen activity of the original malware. The pseudo-code in listing 2 shows how KillerScreenshot ensure to capture the right screenshot corresponding to the malware screen activity. First (line 2 of listing 2), KillerScreenshot waits for the Android Emulator to be ready. Afterwards, the program install and execute the original malware and makes sure that the malware generates an activity on the screen (line 5-15 of listing 2). To do so, KillerScreenshot compares (using SSIM with a threshold empirically chose at 0.95) screenshots taken at regular intervals to the original home screen of the emulator until a screenshot is distant enough. Then, the program must wait for the ongoing motion on screen to finish before taking a final screenshot of the malware screen activity (line 16-24 of listing 2). This is required because displaying an overlay or an activity on screen is not instantaneous and may require up to 2 seconds to completely show up. Finally, when screen seems to be stabilized, KillerScreenshot returns the last screenshot taken (line 16-24 of listing 2).

Adversarial example verification

Once KillerScreenshot has successfully collected a witness screenshot from the original malware, it starts testing adversarial examples. In a very similar way to the procedure described in the listing 2, KillerScreenshot installs and execute the adversarial example on the emulator and take screenshots at regular intervals (every 100 milliseconds). The program compares each screenshot with the witness screenshot obtained during context initialisation until the similarity value is above a certain threshold (chosen at 0.95) or the timeout is reached. If both screenshots are identical, KillerScreenshot successfully asserted the AE as functional, and otherwise, the AE is considered faulty.

Algorithm 2 KillerScreenshot: Context initialisation cycle

Require:
apk
avdReady ← *isAvdReady*()
while not *avdReady* **do**
    avdReady ← *isAvdReady*()
avd ← *getRunningAvd*()
avd.installApk(*apk*)
*ssim*1 ← 1
timeout ← 1000
startTime ← *getTime*()
homeScreenshot ← *avd.takeScreenshot*()
tempScreenshot ← *null*
avd.execute(*apk*)
while *ssim*1 > 0.95 and *getTime*() − *startTime* < *timeout* **do** ▷ Checking that the apk produces a screen activity
    tempScreenshot ← *avd.takeScreenshot*()
    *ssim*1 ← *getSsim*(*homeScreeshot*, *tempScreenshot*)
    wait(100)
if *ssim*1 < 0.95 **then**
    startTime ← *getTime*()
    finalScreeshot ← *avd.takeScreenshot*()
    *ssim*2 ← *getSsim*(*tempScreenshot*, *finalScreenshot*)
    while *ssim*2 <= 0.95 and *getTime*() − *startTime* < *timeout* **do** ▷ Wait for apk screen activity to stabilize
        newScreenshot ← *avd.takeScreenshot*()
        *ssim*2 ← *getSsim*(*finalScreenshot*, *newScreenshot*)
        finalScreenshot ← *newScreeshot*()
        wait(100)
    if *ssim*2 >= 0.95 **then**
        return *finalScreenshot*
return *null*

5.3.2 Evaluation

Implementation details

Similarly to KillerTotal (see section 5.2), we uses Nomad as workload orchestrator to deploy multiple KillerScreenshot instances on to of our infrastructure. KillerScreenshot is made of one python program and one Android emulator, running into a Docker container. As for KillerTotal, the python program is responsible for listening on a queue to get new apks to test on the Emulator. On average, one KillerScreenshot instance takes 6.1 second to test a valid sample and a timeout is set at 30 seconds. Therefore, the time to process an entire dataset of adversarial examples depends on the number of functionnal samples in the dataset. Our infrastructure can run 62 KillerScreenshot instances in parallel, thus it takes between 20 and 100 minutes to test a dataset of 12096 applications.

Experimental results

Dataset	Error Rate	Samples tested
D_1^m	1.99	12096
D_2^m	1.28	12096
D_3^m	0.64	12096
D_3^m	0.95	12096

Table 5.4 – Error rate calculated by KillerScreenshot for all datasets generated by KillerDroid

To evaluate the capacity of KillerDroid (see section 4.2) to produce functional adversarial examples, we submitted all generated samples to KillerScreenshot. In total, KillerDroid produced 4 datasets of 12096 malware variants. The results obtained by KillerScreenshot are reported in table 5.4. We notice that KillerScreenshot has been able to successfully test all samples from the four datasets. More importantly, KillerScreenshot helped to demonstrate that KillerDroid is able to produce functional adversarial example with an error rate below 2%.

5.4 Conclusion

With KillerTotal, we proposed a scalable platform to evaluate the performance of fully-fledged antivirus products, available to the public on the Google Play Store. The proposal demonstrates the ability of KillerTotal to scale infinitely horizontally, thus allowing for a rapid test of thousands of application samples at a high pace. Many works in the research community consider VirusTotal as the most reliable platform to validate malware samples and create ground truth datasets. However, thanks to KillerTotal, we have been able to prove that fully-fledged antivirus products are way more efficient than versions of antivirus aggregated on VirusTotal. By improving KillerTotal with new drivers to support more antivirus, our approach could supplement VirusTotal and help build ground truth datasets of higher quality with more confidence. Furthermore, we have shown that although KillerTotal is more effective than VirusTotal, none of the 7 antivirus tested were effective against adversarial examples generated by KillerDroid. Such findings confirm that most advanced antivirus products on the market are unable to deal with the newest attacks used by malware authors to bypass detection.

In addition, we proposed KillerScreenshot, an approach to automatically verify if synthetically crafted malware variants are fully working. KillerScreenshot complements the work done with KillerDroid by enabling research to build challenging datasets to improve malware detection systems. By combining the two approaches, we can not only create malware samples that use the most advanced evasion techniques, but also verify that the behavior of these variant has not been altered during manipulation.

We hope that KillerTotal will help the research community test Android applications with more confidence with the most advanced antivirus version publicly available. We think about releasing KillerTotal as an open source project, as the approach would greatly benefit from the Android security community to support more antivirus products. Similarly, considering the many recent works on adversarial attack in the Android community, we believe that KillerScreenshot is a promising proof of concept that can greatly help researchers to craft high quality malware variants.

Chapter 6

Future works

The promising results found during the work carried out in this thesis constitute an additional step toward improving the Android ecosystem's global security. However, attackers are constantly moving forward, finding new methods to bypass security protections put in place by all actors in the ecosystem. This forces ecosystem actors, both from academia and industry, to improve existing protection systems and find new solutions to anticipate new attacks to come. In this chapter, we first present our short term vision for the work achieved in this thesis. Then, we propose long term perspectives that could contribute to an Android ecosystem safer for all users.

6.1 Short term future works

With the work carried out during this thesis, we sought to improve existing malware detection systems. The approaches we proposed have proven to be effective against the current malware landscape but could quickly become obsolete given the rapid evolution of new attack techniques. In this section, we first present the incremental improvements that our work could benefit from to keep the pace against new malware threats. Then, we present new research directions that could be followed from approaches presented in this dissertation.

6.1.1 Towards more diversified malware datasets

So far, KillerDroid (see section 4.2) allowed for the production of more than 50 000 malware variants. From a couple of one benign and one malware application, our toolchain can quickly generate 12 096 variants of the original malware, built using a combination of recent evasion techniques, such as *dynamic code loading* and *bytecode encryption*. Our evaluation showed that malware variants produced by KillerDroid are already capable of evading state-of-the-art malware detection methods, both from academia and the

industry. KillerDroid is already able to produce a wide variety of malware variants by combining various evasion techniques. To keep up with the rapid development of the malware landscape, KillerDroid is implemented with a modular architecture which makes it simple to add new evasion techniques in order to diversify even more the possible obfuscation tactics when generating malware variants. Such a modular architecture will help cover more and more corner cases in generated datasets. We believe that the research community would greatly benefit from enlarged and more diversified datasets than those currently offered with KillerDroid. As future works, we have identified new evasion attacks used by existing malware found by research works and envision to add them as new mutations in the corpus of *tactics* used by KillerDroid.

New evasion attacks

Remote code loading. Currently, KillerDroid generates malware variants by dynamically loading the malicious payload two different local locations: the assets directory, and the *host* application. Recent works [137] show that attackers are creating malware that use *remote code loading* to execute a malware payload downloaded from a remote location, such as the attacker's server. Such practices makes it harder to detect a malicious behviour with only static analysis as the malicious payload does not exit in the application file before its execution. Such remote code loading evasion technique could be implemented as a new tactic in the KillerDroid toolchain to increase the number of malicious payload dissimulation options.

Leveraging side channel attacks. As a future work, we envision making malware variants generated by KillerDroid stealthier by leveraging recent works [49] that uncovered several side channel attacks to access privacy sensitive data without the explicit permission to do so. As previously demonstrated, permissions requested by an Android application are particularly relevant in detecting suspicious behaviours. However, malware often need to request dangerous permissions to correctly execute their malicious behaviours. A recent study [49] has shown that recent applications, both malicious and legitimate manage to bypass these permission requirements by leveraging side channel attacks. For example, applications can infer the device's unique identifier, or the device's MAC address without requesting the related permission by leveraging unprotected native IOCTL calls in native code. A new version of KillerDroid could implement these side channel attacks to provide malware variants with an API to get access to the data or the

device's features they need without having to declare the related suspicious permissions in the variant's manifest.

Challenging dynamic analysis systems.

As of today, evasion attacks implemented in the KillerDroid toolchain are mainly focused on challenging malware detection systems that leverage a static analysis approach to analyse applications. Drebin [21] and MaMaDroid [19] are two malware detection frameworks based on static analysis and the response time of most antivirus products on VirusTotal [42] suggests that most engines are also based on static analysis. As such, we have currently no evidence that malware variants produced by KillerDroid could effectively challenge malware detection approaches based on dynamic analysis [137, 209].

We therefore envision adding two novel tactics to KillerDroid that would allow the generation of new malware variants specifically designed to challenge malware detection systems based on dynamic analysis.

1. **Deferred malicious payload execution.** The first strategy, namely *deferred malicious payload execution*, consists in making malware variants created by KillerDroid able to defer the execution time of their malicious payload. Currently, the bootstrap sequence of KillerDroid is implemented in such a way that the malicious payload of a variant will always be executed right after its launch by the user. This approach could easily be detected by a dynamic analysis tool as starting the application is enough to trigger the malicious behaviour. Similar to *time bombs* [210] already used by attackers in their malware, *deferred malware payload execution* will allow KillerDroid generated malware variants to load and execute their malicious payload after a certain amount of time, or at specific hours.
2. **Conditional malicious payload execution.** The second strategy envisioned is name *conditional malware payload execution* and will allow new KillerDroid generated variants to load and execute the malicious payload under certain conditions. Since the payload of the variant is contained in a legitimate application, this would allow the variant to behave legitimately until the condition to run the payload is met. For example, the payload execution could occur only when the user connects its device to a Wi-Fi network, or when the user executes a legitimate action with the variant, like sharing a photo. Similarly to *logic bombs* [210], *conditional malware payload execution* will be likely to put dynamic analysis scanners in difficulty as it could be hard to meet the conditions required to trigger the

malicious payload execution in an automated fashion.

6.1.2 Future of machine learning malware detection

Today, there are more and more studies that leverage machine learning to improve the performance of malware detection in the Android ecosystem. Over time, machine learning approaches and conditions used to evaluate them became very heterogeneous. First, studies design detection systems based on many different learning algorithms, ranging from linear to deep neural network classifiers. The data used for training these classifiers have various origins and systematically differ from one study to another. Moreover, as already suggested in the past [23], there are no convention in the research domain to decide whether an application sample is legitimate or malicious. Besides, the emergence of studies that leverage adversarial attacks offers various possibilities to evaluate machine learning-based detection systems. Recent works have provided multiple ways to create adversarial examples with various obfuscation and evasion techniques, which increase even more the data that can be used to evaluate detection systems.

Taking machine learning-based detection benchmarks to the next level

Such heterogeneity in ML-based malware detection emphasizes the need for more systematic and methodical ways to benchmark the performance of machine learning malware detection approaches. With DroidAutoML, we proposed an approach that partly solve the problem with a framework that can quickly test several learning algorithms and hyper-parameter combinations in order to find the best possible configuration to maximise the detection rate of a given machine learning detection approach. However, currently, the choice of the data used to train and test algorithms is left to the user. Moreover, DroidAutoML can only benchmark four learning algorithms.

As a future work, we will explore the possibility of designing a more general and exhaustive protocol to benchmark machine learning-based detection systems. By reusing the concepts presented with DroidAutoML, we envision a platform with the necessary abstractions to allow for a benchmark of detection systems with more criteria, in a more controlled manner. First, the platform could allow training and testing classifiers with various but yet fixed datasets. Thus, it would be possible to evaluate the robustness of a classifier to datasets created with various adversarial attacks, like the one presented with KillerDroid (see section 4.2). Secondly, DroidAutoML would benefit from support-

ing more learning algorithms and more feature extraction approaches.

To improve the efficiency of DroidAutoML, we are currently looking at replacing the *grid search* approach by a *bayesian optimization*. As more learning algorithms will be added to DroidAutoML, framework performances will become critical. Currently, the search for the best combination is done by performing a brute force approach (i.e. a grid search) among all possible combinations of hyper-parameters. Thus, the computation resources required to find the best solution grow significantly with the number of hyper-parameter combinations to test. Unlike a grid search approach, a *bayesian optimization* does not require trying all the hyper-parameter combinations. Instead, the *bayesian optimization* searches along the space of hyper-parameters by learning as it tries them. Implementing such an approach could greatly improve the speed of DroidAutoML as it reduces the number of hyper-parameter combinations to test for all the learning algorithms tested.

Improving malware detection using parallel machine learning classifiers

To further improve the accuracy of machine learning-based malware detection scanners on Android, we envision taking advantage of KillerDroid's ability to generate malware variants to boost the training of parallel machine learning classifiers.

In recent years, to fight against the proliferation of new heterogeneous malware families, studies [211, 212] have investigated malware detection through multiple machine learning classifiers. In particular, Google [213] patented a similar approach but has not publicly released it yet. Such approaches rely on a composite classification models, i.e. a combination of heterogeneous classifiers based on multiple learning algorithms. Classifiers are trained using diverse malware datasets and different set of features extracted either statically or dynamically from samples. Thereafter, trained classifiers are used in parallel to predict the maliciousness of a sample and a combination function is responsible for gathering classifiers' predictions and issuing a final result.

Such an approach can be used to build multiple classifiers specifically trained to detect a particular malware family or a particular malicious behaviour. Instead of relying on a single classifier trained to generalize the problem of malware detection globally, these classifiers can therefore be chained or used in parallel to make predictions. One advantage of this approach is that each classifier can be trained with a training dataset and a set of features tailored to detect a specific malware family or behaviour. As new malware

families are discovered, such systems can therefore be improved by adding new classifiers in the chain.

However, one limitation of this approach is the need for a significant number of malware samples to correctly train each classifier. Splitting existing public malware datasets into subsets of malware families will significantly decrease the number of samples available for one classifier. To solve the aforementioned, we envision using KillerDroid to craft a significant number of malware variants for each malware family. So far, KillerDroid has been able to generate more than 50 000 malware variants, mostly using malware from the *Ransomware* family. To diversify the malware variants that KillerDroid can create, we plan on improving the KillerDroid toolchain to make it support many more possible mutations. For example, currently, KillerDroid is only able to create malware variants where the malicious payload is immediately triggered when the application is launched. A novel mutation approach could be to craft malware variants where the malicious payload is triggered either after a certain amount of time, or when a certain condition is met, such as a system event or a specific user action.

By diversifying both the kind of malware used to craft variants and the number of mutations supported by the KillerDroid toolchain, we could be able to create datasets sufficiently large to efficiently train a parallel machine learning classifier. Thus, we further aim at building a new classifier based on a multilevel architecture that uses both linear and ensemble learning algorithms to train a sub-classifier for each malware family represented in previously generated datasets.

6.1.3 Poisoning VirusTotal

As a future work, we envision a novel attack which allows an attacker to irreversibly corrupt and subvert detection results of antivirus engines running on aggregators such as VirusTotal. With the help of KillerDroid, which is able to quickly generate thousands of adversarial examples that look like a chosen legitimate application, we would show that it is possible to influence detection results of files scanned in the past by uploading carefully chosen adversarial examples on the aggregator. Such an attack will allow antivirus vendors to adapt their detection mechanisms to prevent the increase of false positives (i.e. legitimate samples wrongly detected as malicious) in their system.

As discussed in section 5.1, many research works and security companies still rely on VirusTotal to determine whether a file sample is malicious or benign. Such centralized platforms bring freshness to datasets which is necessary to retrain models and update

antivirus databases and keep acceptable detection performance over time. Unfortunately, crowdourcing based solutions are known to suffer from byzantine behaviours as there is no control on the incoming data. There is little information on how companies enrolled in the VirusTotal program review the stream of incoming files acquired through user submissions. While some unseen malware variants may be automatically detected as malicious by some engines, there is probably a lot of human intelligence involved to double check suspicious samples and effectively label them as malware. An attacker may intentionally feed such crowdsourcing platforms with adversarial examples specifically crafted to trick scanners that rely on the input data to perform malware detection. Regarding VirusTotal, a major point of failure is that to improve their software, antivirus vendors share together newly detected samples across their products to share the knowledge and improve their detection performance. Thus, if an attacker manages to generate detection errors on one antivirus, it is a safe bet that the error will spread globally among all the antivirus aggregated on VirusTotal. Several research studies have already highlighted attacks on crowdsourcing based systems [214, 215] showing that such attacks are efficient to disturb decision-making algorithms relying on the input data. In the field of malware and unwanted software however, no such study has been carried out.

When carrying out our experiments on KillerDroid (see section 4.2), we empirically noticed that we were able to influence scan results of antivirus on the aggregator. By generating malware variants specifically crafted to look like existing benign legitimate Android applications, we have been able to show that after a sufficient number of submissions on VirusTotal, most scanners start considering in turn the original legitimate application as malware. More precisely, this attack makes antivirus products believe that an application they considered benign is actually a malware. In a real case example, this attack can be seen as a denial of service against the mobile application of a business company. To gain market share in a business competition, a company could carry out this attack on another company's application to make it look like a malware on the VirusTotal platform. Consequently, all antivirus engines running both on VirusTotal and on end devices will warn users that the application is a malware until users ultimately no longer trust the attacked company.

6.2 Long term future works

In this section, we envision two complementary perspectives to improve the security of the Android ecosystem. We first propose a shift towards a more collaborative model to deal with malware detection. Then, we explain how moving mobile applications to the cloud could improve the Android security platform.

6.2.1 Towards more collaborative efforts

To move towards a safer Android ecosystem, we envision more collaboration between actors within it. Currently, the Android ecosystem largely depends on Google's efforts to guarantee security. The *Google Play Store* mainly relies on its homemade solution, *Bouncer* [216], to prevent malicious applications from being published on the platform.

A new model, built on top of collaborative means could be imagined to ease the collaboration between security actors in the ecosystem. Such a model could propose an abstracted API that would allow various entities, including commercial antivirus companies and research institutes to contribute to the evaluation of applications uploaded to multiple stores. The premises of such an API have been successfully demonstrated by the VirusTotal platform [42], where an alliance of more than 60 antivirus vendors can scan millions of files uploaded on a single platform. Such an API could follow an open source model, jointly developed and maintained by an alliance of stores, antivirus vendors and researchers. This collaborative approach would benefit to every parties. Stores could use the API to verify uploaded applications on both commercial antivirus and implementations of multiple research projects. In addition, antivirus vendors and research institutes could profit from data sent on their systems by stores to increase their security-related datasets and improve their malware detection models.

6.2.2 Moving applications to the cloud

One idea to improve Android security would be to move applications to the cloud instead of asking users to install them. The increase of smart devices' computation resources as well as more efficient and reliable networks now makes it possible to move applications to the cloud. Cloud mobile applications are applications that run on servers external to the mobile device and are accessed over the Internet with a browser. For example, Google recently launched *Stadia* for smartphones [217], a cloud gaming service that allows users

to stream video games directly on their smartphone. Instead of executing the game locally on the smartphone, the game is executed on a remote server and image frames are sent to the user's device over the network.

In terms of security, cloud mobile applications could help secure smart devices by removing the need for users to install applications on their device. As cloud applications can be accessed in a browser, only a few verified applications could be allowed to be installed on the user's smart device. This would make securing the Android platform easier, as it would drastically reduce the attack surface and the number of applications to verify.

While this mobile cloud application model can help secure the Android platform, it does not solve the malware problem and only shifts it towards cloud providers. Indeed, there will always be a store to allow developers to offer cloud applications to users. Attackers will still be able to distribute cloud mobile malware. However, it will be harder for attacker to escape the sandbox provided by mobile browsers.

Conclusion

Nowadays, with the advent of smart devices in our everyday life, smart ecosystems are threatened by bad actors that illegitimately pervade the system with malicious applications, namely malware, to cash in on smart device users. As the number of infected devices continues to rise, the fight against malware has become of the uttermost importance for all legitimate contributors of smart ecosystems. Despite the multiple security locks implemented by software platforms and stores to detect malware, attackers use ever smarter techniques to build malware that evade detection. This situation demonstrate the need to improve and adapt existing malware detection systems so they can deal with the next generation of malware.

6.3 Summary of contributions

In this thesis, we have proposed four contributions that seek to progress toward more robust and more intelligent Android malware detection systems:

DroidAutoML. In chapter 3, we presented DroidAutoML, a generic and scalable approach based on a microservice architecture that allow for an automatic increasing of the performance of existing Android machine learning-based malware detection techniques. From two inputs, a malware dataset and a feature extraction method, DroidAutoML automatically performs an extensive and exhaustive search by training various learning algorithms with thousands hyper-parameters combinations to find the highest possible malware detection rate. DroidAutoML successfully increased the accuracy of two existing state-of-the-art machine learning detection approaches while not being too expensive in terms of time and resources consumed. This contribution is based on the work presented in the following paper:

— Yérom-David Bromberg and Louison Gitzinger, « DroidAutoML: A Microservice Architecture to Automate the Evaluation of Android Machine Learning Detection Systems », *in*: *IFIP International Conference on Distributed Applications and Interoperable Systems*, Springer, 2020, pp. 148–165 [218]

Groom. We further presented GROOM, a fast static analysis approach which extracts a novel efficient feature set from Android applications to improve machine learning detection. In particular, GROOM extracts specific features that enable detection systems to take into account the use of recent obfuscation techniques such as reflection, native code execution, dynamic code loading and encryption. GROOM outperformed two state-of-the-art machine learning approaches in terms of accuracy while being faster in analysing applications to extract required features.

KillerDroid. In chapter 4, we proposed KILLERDROID, an approach that enables to craft challenging malware variants with the aim of diversifying malware datasets to evaluate detection scanners. KILLERDROID crafts new malware variants by weaving existing malware applications into benign applications using advanced evasion techniques such as *dynamic code loading* or *native code execution*. State-of-the-art scanners from both academia and the industry have proven to be vulnerable to adversarial datasets created by KILLERDROID. Datasets created by KILLERDROID further helped highlight the vulnerabilities of scanners to specific evasion techniques.

KillerTotal. In chapter 5, we presented KILLERTOTAL, a large-scale deployment platform to evaluate individually the publicly available versions of the seven most efficient Android mobile antivirus products available on the Google Play Store. By deploying multiple instances of antivirus in parallel, KILLERTOTAL can submit datasets of thousands of applications to calculate the malware detection rate of these antivirus. KILLERTOTAL allowed us to demonstrate that publicly available versions of antivirus outperform versions provided in antivirus aggregators. Results obtained suggest that KILLERTOTAL could replace commonly used aggregators like VIRUSTOTAL to build ground truth datasets with more accuracy. KILLERTOTAL further demonstrated that mobile antivirus products remain inefficient against datasets generated by KILLERDROID.

KillerScreenshot. In chapter 5, we further presented KILLERSCREENSHOT, an approach to automatically vet the correct behaviour of synthetically crafted malware. KILLERSCREENSHOT complements the work achieved by KILLERDROID by automatically verifying that adversarial examples generated by KILLERDROID are correctly triggering their malicious behaviour at runtime. By ensuring the proper execution of malware variantes

generated, KillerScreenshot contributes to the increase in quality of malware datasets generated by adversarial attacks in order to make these attacks more *realistic* and challenging for existing malware scanners.

Overall, our work has highlighted several weaknesses of existing Android malware detection systems and proposed concepts and tools to improve them.

This is however, a small step toward making smart ecosystems a safer place. The rising number of smart devices make such ecosystems an inexhaustible business for attackers. Can malware detection ever become robust enough to protect users in smart ecosystems against malware infection without hindering them? We recognize that, a determined attacker will always find ways to dissimulate a malicious behaviour and make it undetectable to any detection scanner. The only way to prevent such attacks would be to raise security barriers to such an extent that it would be destructive for legitimate businesses and user experience. In a nutshell, tension between security and liberty must be thoroughly studied. We conclude this thesis with the hope that the presented contributions will provide the Android research community useful insights on how Android malware detection can be improved while keeping the ecosystem useful for everyone.

List of publications

— Yérom-David Bromberg and Louison Gitzinger, « DroidAutoML: A Microservice Architecture to Automate the Evaluation of Android Machine Learning Detection Systems », *in*: *IFIP International Conference on Distributed Applications and Interoperable Systems*, Springer, 2020, pp. 148–165 [218]

Chapter 7

Résumé en français

7.1 Contexte

De nos jours, nous sommes entourés par une flotte de périphériques intelligents, connectés entre eux par différents protocoles réseaux. Ces périphériques, tels que les smartphones, les télévisions connectées ou les montres connectées, sont qualifiés d'intelligents parce qu'ils fonctionnent ensemble de manière autonome et interactive pour améliorer notre niveau de vie. Ainsi, votre smartphone peut vous notifier au travail lorsque que votre caméra connectée détecte que vos enfants sont bien rentrés à la maison. Lorsque vous planifiez un itinéraire en voiture, votre voiture connectée saura automatiquement comment vous guider jusqu'à votre destination. Pour offrir ces services, les périphériques que nous utilisons tous les jours font partie d'écosystèmes plus larges, dans lesquels plusieurs entreprises collaborent. D'un point de vue économique, les écosystèmes peuvent être vus comme une inter-dépendance entre l'offre, c'est-à-dire les entreprises qui vendent des périphériques et des services, et la demande: les utilisateurs. Dans ces écosystèmes, les entreprises doivent garantir une certaine qualité de service pour assurer une bonne rétention des utilisateurs et ainsi être rentable.

Dans ce contexte, des entreprises comme Google et Apple offrent un écosystème fort et cohérent en organisant leur activité autour d'une plateforme logicielle, comme Android ou IOS. Ces plateformes s'exécutent sur les périphériques et permettent à l'utilisateur d'y installer des programmes appelés *applications*. Les applications sont des programmes individuels qui offrent à l'utilisateur des services variés pour être productif (envoyer des mails ou des SMSs, gérer son agenda, ...), pour se divertir (réseaux sociaux, vidéos, ...) ou encore pour jouer. Les plateformes logicielles proposent aussi un kit de développement logiciel (Software Development Kit, SDK) qui encouragent et facilitent le développement d'applications compatibles avec la plateforme. Les SDKs ouvrent l'écosystème à la communauté de développeurs et permettent à des entreprises tierces (comme Facebook, Airbnb, Uber, ...) de concevoir des applications pour les utilisateurs de périphériques.

Une fois installées, les applications utilisent les capteurs intégrés aux appareils (camera, détecteur d'empreinte, accéléromètre, localisation GPS, ...) pour améliorer en continu la qualité des services proposés. Avec les moyens de communication de l'appareil (LTE, WiFi, Bluetooth, ...), les applications peuvent envoyer les données collectées à n'importe quel serveur tiers, permettant ainsi aux développeurs de récolter des retours précieux sur les usages des utilisateurs. En échange de ces données, les utilisateurs peuvent profiter d'applications plus intelligentes, avec des services de qualité comme la publicité ciblée, les commandes rapides ou encore le paiement en un seul clic.

Pour faciliter l'accès aux applications créées par les développeurs, des services en lignes, appelés *stores*, offrent à l'utilisateur un espace centralisé pour les chercher et les télécharger. À l'instar d'un marché, les *stores* proposent aux vendeurs (les développeurs) de téléverser, publier et faire de la publicité leurs applications sur le service. Pour aider les utilisateurs à naviguer parmi les millions d'applications proposées, les *stores* les organisent en catégories, comme *Art et culture*, *Divertissement*, *Jeux*, etc.

Avec le temps, le progrès technologique a permis à ces écosystèmes de prospérer tout en se révélant rentables pour tous les acteurs. Ce cercle vertueux s'est traduit par une augmentation substantielle du marché des appareils intelligents. Les plateformes logicielles, dominées par Android et Apple IOS qui détiennent ensemble 98% du marché, comptent des milliards d'appareils actifs dans le monde. Aujourd'hui, Android est installé sur 2.5 milliards d'appareils actifs dans le monde. En comparaison, la plateforme IOS est installée sur 1.4 milliards d'appareils. Suivant cette tendance, les *stores* ont eux aussi observé une croissante formidable. En dix ans, le nombre d'applications publiées sur les deux *stores* principaux (le Google Play Store et l'App Store d'Apple) est passé de quelques milliers à plusieurs millions.

Pour rester rentable dans ce contexte, les écosystèmes intelligents doivent gagner la confiance des utilisateurs. Cependant, comme n'importe qui est autorisé à publier de nouvelles applications, les *stores* doivent vérifier que les applications publiées sur leur plateforme sont de bonnes qualité et ne comportent pas de risque pour l'utilisateur. Malheureusement, la croissance de ces écosystèmes a incité des personnes mal intentionnées (des attaquants) à profiter du système illégalement.

7.2 Vulnérabilité de l'écosystème Android

Les attaquants profitent à tous les niveaux des vulnérabilités de l'écosystème pour cibler les utilisateurs avec des logiciels malveillants, nommés malwares (pour *malicious software*), qui exploitent les données de l'utilisateur à son insu pour l'espionner ou se faire de l'argent sur son dos. Dans ce contexte, en raison de sa popularité, l'écosystème Android est ciblé par 98% des attaques sur les appareils intelligents. Pour exploiter les failles de l'écosystème, les attaquant créent différents types de malwares, qui ont été catégorisés en familles :

— Au niveau du système d'exploitation : les attaquants peuvent exploiter des vulnérabilités pour déployer des *rootkits*, pour obtenir un accès *root* sur l'appareil ciblé.

— Au niveau du *framework* : les malwares peuvent demander la permission d'accéder à des données sensibles de l'utilisateur telles que sa localisation GPS ou sa liste de contacts. Les malwares peuvent aussi demander au *framework* la permission d'effectuer des actions dangereuses telles que supprimer les fichiers de l'utilisateur ou prendre des photos.

— Au niveau applicatif : les malwares peuvent exploiter d'autres applications vulnérables installées sur l'appareil, par exemple pour arrêter l'application ou accéder à des informations sensibles manipulées par l'application ciblée.

— Au niveau du réseau : les malwares peuvent effectuer des dénis de service sur d'autres applications, envoyer des SMSs ou passer des appels téléphoniques à des numéros surtaxés.

Cette croissance inquiétante de la menace des malwares sur l'écosystème soulève un défi de taille : comment garder un écosystème ouvert aux contributeurs tierces tout en garantissant la sécurité des utilisateurs ?

7.3 Solutions existantes

Pour contrecarrer la menace induite par les malwares, les contributeurs travaillant sur la sécurité d'Android, provenant à la fois de la recherche et de l'industrie, renforcent la sécurité de l'écosystème avec de nouvelles défenses. La plateforme Android se sert des capacités du kernel Linux tel que le *sandboxing* d'applications, *SELinux* et le démarrage sécurisé (*verified boot*) pour mieux isoler les applications entre elles et mieux controller

l'accès aux données des applications. Avec son model de permissions, qui a été sujets à de nombreuses améliorations ces dernières années, la plateforme Android demande explicitement aux applications de déclarer les fonctionnalités dont elle a besoin pour fonctionner correctement. Toute tentative d'une application d'accéder à des ressources sans y être autorisée sera refusée par le système. Ainsi, pour accéder à des fonctionnalités telles que l'appareil photo, les appels téléphoniques ou les sms/mms, une application doit explicitement demander la permission à l'utilisateur.

Pour freiner l'infection des appareils en amont, les *stores* ont mis en place des systèmes de validation qui permettent de filtrer les applications soumises sur leur plateforme. Ces processus de validation, comme Google Play Protect, utilisent à la fois des techniques manuelles et automatiques pour analyser le flux d'applications téléversées et vérifier qu'elles ne sont pas malveillantes. Comme ces protections ne sont pas suffisantes pour supprimer tous les malwares, des entreprises commerciales ont développé des antivirus à installer sur l'appareil qui analysent les applications lors de leur installation. Par ailleurs, le domaine de la recherche a montré des résultats prometteurs avec des approches innovantes pour améliorer la détections de malwares sur la plateforme Android.

Pour garantir la légitimé d'une application, il est nécessaire de collecter des données qui peuvent renseigner sur ses intentions et son comportement. Pour cela, deux techniques d'analyses logicielles sont principalement utilisées : l'analyse statique et l'analyse statique, qui permet de rapidement analyser les fichiers binaires et les resources d'une application sans avoir à l'exécuter, et l'analyse dynamique, qui permet d'analyser l'application en l'exécutant pour étudier son comportement.

Ensuite, les systèmes de détection utilisent un oracle pour décider automatiquement si une application doit être considérée comme malveillante ou légitime. Les oracles sont des systèmes de décision qui font un choix, à partir des données récupérées lors de l'analyse de l'application. Les oracles sont en général basés sur deux stratégies de détection différentes :

— détection par signature : avec la détection par signature, l'oracle prend sa décision en comparant des informations comme la signature d'un fichier ou certains motifs dans le code binaire de l'application avec un ensemble d'heuristiques et de règles générées à partir de logiciels malveillant déjà étudiés auparavant.
— détection par machine learning : avec la détection par machine learning, les oracles utilisent une base de données de malwares existants pour entrainer un classificateur

7.4 Limites des solutions existantes dans l'écosystème

Malgré les solutions mises en place, une étude récente a montré qu'au moins 10 000 malwares appartenant à 56 familles différentes ont été publié sur le Google Play Store entre 2014 et 2018. De plus, les rapports de sécurité d'entreprises spécialisées montrent que le nombre d'appareils Android infectés continu d'augmenter en 2020. En réalité, les attaquants deviennent plus intelligents avec le temps et utilisent des techniques de plus en plus sophistiquées pour vaincre les défenses mises en place par tous les acteurs de l'écosystème Android.

Pour vaincre les méthodes d'analyse statique, les attaquants utilisent des techniques d'évasion avancées telles que le chargement dynamique de code, l'utilisation de la *reflection* ou encore le chiffrement de bytecode pour cacher le code malveillant dans l'application et le rendre inatteignable pour les programmes d'analyse statique. Concernant l'analyse dynamique, les systèmes de détections exécutent souvent les applications dans des machines virtuelles plutôt que sur des vrais appareils pour permettre l'analyse d'applications à grande échelle. Cependant, Petsas et al. [28] ont montré que les malwares sont en mesure de détecter lorsqu'ils sont exécutés dans un environnement virtuel. Pour vaincre la détection par signature, les attaquants utilisent le *polymorphisme* et différentes techniques d'offuscation qui permettent de changer complètement la structure du code d'une application sans affecter sa sémantique. Cela permet à l'attaquant de rapidement créer des variantes d'un malware avec une signature jamais vue auparavant. Le machine learning (ML) est massivement adopté pour essayer de contrer ce phénomène. Cependant, les récentes avancées à propos de la détection de malwares par machine learning sont sujettes à plusieurs limitations, que nous résumons ci-dessous.

Limites des solutions actuelles. Bien qu'il s'agisse d'une première étape vers une amélioration de la détection, la plupart des études utilisant le machine learning négligent l'affinage des algorithmes d'apprentissage. Les performances des classificateurs dépendent de plusieurs aspects : l'algorithme d'apprentissage utilisé, les paramètres (hyper-paramètres) utilisés pour configurer l'algorithme et la recherche de features. Jusqu'à maintenant, seuls quelques essais ont été mené pour tenter d'optimiser les approches de détection de malwares basées sur du machine learning. Les évaluations de ces approches sont pour la plupart effectuées empiriquement, avec un processus manuel en utilisant peu de combinaisons d'hyper-paramètres ou celles par défaut. De plus, les features extraites par les approches proposées ne permettent pas de tenir compte des techniques d'évasion

récemment employées par les malwares telles que le chargement dynamique de code ou le chiffrement de bytecode.

Limites des datasets de malwares publics. Des études récentes ont démontré que les modèles existants de classification basés sur le machine learning peuvent être vaincu par des malwares spécifiquement conçus dans ce but. Ces études pointent du doigt que les datasets de malwares employés pour entrainer les classificateurs ne sont pas assez hétérogènes et manquent d'échantillons de malwares diversifiés. Pour entrainer un classificateur fiable, les datasets de malware doivent contenir une large variété d'échantillons qui permettront au modèle de mieux généraliser le problème et de détecter plus de cas particuliers. Spécifiquement, les datasets de malwares existants manquent d'échantillons qui utilisent des techniques d'évasion avancées comme le chargement dynamique de code ou le chiffrement de bytecode.

Limites des méthodes de vérification des échantillons de malwares. La création de datasets de vérité de terrain de qualité est une étape fondamentale pour implémenter et évaluer de nouvelles méthodes de détection de malwares. L'efficacité d'un système de détection est directement lié avec la qualité du dataset utilisé pour entrainer et tester celui-ci. Cependant, la qualité du dataset de vérité de terrain peu être limitée part (i) des échantillons mal labellisés, c'est-à-dire des échantillons légitimes labellisés comme malveillants ou *vice versa*, et (ii) les échantillons non réalistes, c'est-à-dire dépassés et/ou non-fonctionnels.

La grande majorité des approches basées sur le machine learning se fient aux agrégateurs d'antivirus pour déterminer si une application est légitime ou malveillante et ainsi construire un dataset d'entrainement. Les agrégateurs d'antivirus, comme VirusTotal sont des plateformes en ligne qui exécutent les produits de multiples entreprises de sécurité pour examiner les fichiers téléversés par les utilisateurs. Toutefois, il a été suggéré que ces agrégateurs pourraient introduire un biais dans les datasets conçus avec cette méthode, en raison de résultats d'analyse erronés. En effet, pour des raisons techniques et commerciales, les entreprises de sécurité pourraient fournir à l'aggrégateur une version restreinte de leur produit. Cela pourrait artificiellement réduire la précision des agrégateurs comparé à ce qu'il serait possible d'obtenir avec des produits complets.

Les systèmes de détection de malwares actuels dans l'écosystème actuel ont du mal à lutter contre des gens mal-intentionnés de plus en plus intelligents. D'une part, les

solutions traditionnelles qui utilisent la détection par signature ne parviennent pas à efficacement détecter les nouvelles variantes de malwares générées à un rythme élevé par les attaquants. D'autre part, les approches basées sur le machine learning se retrouvent face à des problèmes inhérent du domaine : les modèles doivent constamment être ré-entrainés et optimisés pour être en mesure de continuer à détecter de nouveaux échantillons malveillants. Par ailleurs, le manque de données valides de qualité pour entrainer les modèles empêchent d'obtenir des modèles robustes et efficace.

7.5 Contribution de la thèse

Des efforts supplémentaire sont requis pour résoudre les problèmes susmentionnés. Pour contribuer aux efforts de la communauté de recherche, nous proposons les contributions suivantes :

Automatiser l'évaluation des systèmes de détections basés sur le machine learning. D'abord, nous proposons DroidAutoML, une plateforme basée sur une architecture microservice qui automatise l'évaluation de systèmes de détection de malwares sur Android. DroidAutoML résout automatiquement le problème de sélection de l'algorithme d'apprentissage ainsi que l'optimisation des hyper-paramètres en faisant effet de levier sur *AutoML* pour améliorer la précision des systèmes de détection évalués. DroidAutoML est construit sur une architecture microservice dédiée, spécifiquement conçue pour répondre aux besoins de générécité et de flexibilité requis par le domaine de la détection de malwares sur Android.

Un nouvel ensemble de features pour améliorer la détection. Pour améliorer la précision de détection des systèmes basé sur le machine learning, nous proposons Groom. Groom est un framework qui utilise l'analyse statique pour améliorer sensiblement la qualité des features extraites d'une application, tout en étant plus rapide que les approches existantes. En particulier, Groom extrait des features spécifiques qui permettent au systèmes de détection de capturer des malwares qui utilisent des techniques d'évasion récentes comme l'exécution de code natif, ou le chargement dynamique de code.

Vers une diversification massive des datasets de malwares expérimentaux. Nous proposons KillerDroid, une chaîne d'outils qui permet de fabriquer des vari-

antes de malwares qui utilisent des techniques d'évasion avancées dans le but de créer des échantillons pour diversifier massivement les datasets expérimentaux existants. KillerDroid peut générer des variantes à grande échelle, ce qui permet d'évaluer la robustesse des solutions de détection, venant à la fois du milieu académique et de l'industrie, contre les attaques adversariales.

Améliorer la qualité des datasets de malware sur Android. Enfin, nous proposons deux approches dans le but d'améliorer globalement la qualité des datasets expérimentaux utilisés pour entrainer et tester les systèmes de détection

D'abord, nous proposons KillerTotal, une nouvelle approche pour évaluer individuellement les versions mobiles publiques des 7 antivirus Android les plus efficaces disponibles sur le Google Play Store. KillerTotal a pour objectif de montrer que les versions mobiles de ces antivirus sont plus précises et plus efficaces que les versions exécutées par les agrégateurs comme VirusTotal. Notre système fourni un déploiement à large échelle basé sur l'émulation d'Android dans une machine virtuelle pour automatiser l'initialisation et l'exécution des applications mobile des antivirus évalués.

Ensuite, nous proposons KillerScreenshot, un framework qui permet de certifier que les variantes de malwares produits synthétiquement par des frameworks comme KillerDroid sont fonctionnels. KillerScreenshot vérifie dynamiquement que le comportement d'une variante de malware fabriqué est cohérent avec le comportement du malware original utilisé pour le fabriquer en comparant l'activité à l'écran des deux échantillons.

Bibliography

[1] *Android vs iOS over the past 10 years*, https://mybroadband.co.za/news/software/364290-android-vs-ios-over-the-past-10-years.html, (Accessed on 09/11/2020).

[2] *There are now 2.5 billion active Android devices - The Verge*, https://www.theverge.com/2019/5/7/18528297/google-io-2019-android-devices-play-store-total-number-statistic-keynote, (Accessed on 09/11/2020).

[3] *Apple says there are 1.4 billion active Apple devices - The Verge*, https://www.theverge.com/2019/1/29/18202736/apple-devices-ios-earnings-q1-2019, (Accessed on 09/11/2020).

[4] *Google Play Store: number of apps | Statista*, https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/, (Accessed on 09/11/2020).

[5] • *Number of apps from the Apple App Store 2020 | Statista*, https://www.statista.com/statistics/268251/number-of-apps-in-the-itunes-app-store-since-2008/, (Accessed on 09/11/2020).

[6] Nokia, *Nokia Threat Intelligence Report 2019*, https://pages.nokia.com/T003B6-Threat-Intelligence-Report-2019.html, (Accessed on 07/29/2020).

[7] Yajin Zhou and Xuxian Jiang, « Dissecting android malware: Characterization and evolution », *in*: *2012 IEEE symposium on security and privacy*, IEEE, 2012, pp. 95–109.

[8] *New 'unremovable' xHelper malware has infected 45,000 Android devices | ZDNet*, https://www.zdnet.com/article/new-unremovable-xhelper-malware-has-infected-45000-android-devices/, (Accessed on 09/10/2020).

[9] *This nasty new Android ransomware encrypts your phone – and changes your PIN | ZDNet*, https://www.zdnet.com/article/this-nasty-new-android-ransomware-encrypts-your-phone-and-changes-your-pin/, (Accessed on 09/10/2020).

[10] *EventBot: A New Mobile Banking Trojan is Born*, https://www.cybereason.com/blog/eventbot-a-new-mobile-banking-trojan-is-born, (Accessed on 09/10/2020).

[11] *ExpensiveWall: A dangerous 'packed' malware on Google Play*, https://blog.checkpoint.com/2017/09/14/expensivewall-dangerous-packed-malware-google-play-will-hit-wallet/, (Accessed on 09/10/2020).

[12] *Application Sandbox |Android Open Source Project*, https://source.android.com/security/app-sandbox, (Accessed on 09/10/2020).

[13] *Security-Enhanced Linux in Android | Android Open Source Project*, https://source.android.com/security/selinux, (Accessed on 09/06/2020).

[14] *Verified boot | Android Open Source Project*, https://source.android.com/security/verifiedboot, (Accessed on 09/10/2020).

[15] *Android 6.0 Changes Android Developers*, https://developer.android.com/about/versions/marshmallow/android-6.0-changes, (Accessed on 08/21/2020).

[16] *The number of mobile malware attacks doubles in 2018, as cybercriminals sharpen their distribution strategies | Kaspersky*, https://www.kaspersky.com/about/press-releases/2019_the-number-of-mobile-malware-attacks-doubles-in-2018-as-cybercriminals-sharpen-their-distribution-strategies, (Accessed on 08/22/2020).

[17] *Android app reviews may slow to over a week due to COVID-19 impacts, Google warns developers | TechCrunch*, https://techcrunch.com/2020/03/17/android-app-reviews-may-slow-to-over-a-week-due-to-covid-19-impacts-google-warns-developers/, (Accessed on 08/22/2020).

[18] *Android – Google Play Protect*, https://www.android.com/play-protect/, (Accessed on 08/22/2020).

[19] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini, « MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models », *in*: *Proceedings 2017 Network and Distributed System Security Symposium*, Internet Society, 2017, URL: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/mamadroid-detecting-android-malware-building-markov-chains-behavioral-models/.

[20] Yousra Aafer, Wenliang Du, and Heng Yin, « DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android », *in*: *Security and Privacy in Communication Networks*, ed. by Tanveer Zia, Albert Zomaya, Vijay Varadharajan, and MorleyEditors Mao, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, Springer International Publishing, 2013, pp. 86–103.

[21] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, and Konrad Rieck, « Drebin: Effective and Explainable Detection of Android Malware in Your Pocket », *in*: Internet Society, 2014, URL: https://www.ndss-symposium.org/ndss2014/programme/drebin-effective-and-explainable-detection-android-malware-your-pocket/.

[22] Ambra Demontis, Marco Melis, Battista Biggio, Davide Maiorca, Daniel Arp, Konrad Rieck, Igino Corona, Giorgio Giacinto, and Fabio Roli, « Yes, Machine Learning Can Be More Secure! A Case Study on Android Malware Detection », *in*: *arXiv:1704.08996 [cs]* (Apr. 2017), URL: http://arxiv.org/abs/1704.08996.

[23] Haoyu Wang, Junjun Si, Hao Li, and Yao Guo, « Rmvdroid: towards a reliable android malware dataset with app metadata », *in*: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, IEEE, 2019, pp. 404–408.

[24] *App Annie State of Mobile 2020 Report - App Annie*, https://www.appannie.com/en/go/state-of-mobile-2020/, (Accessed on 08/20/2020).

[25] Mohannad Alhanahnah, Qiben Yan, Hamid Bagheri, Hao Zhou, Yutaka Tsutano, Witawas Srisa-an, and Xiapu Luo, « Detecting Vulnerable Android Inter-App Communication in Dynamically Loaded Code », *in*: *Conference on Computer Communications (INFOCOM)*, 2019.

[26] Davy Landman, Alexander Serebrenik, and Jurgen J Vinju, « Challenges for static analysis of java reflection-literature review and empirical study », *in*: *International Conference on Software Engineering (ICSE)*, IEEE, 2017.

[27] Babak Bashari Rad, Maslin Masrom, and Suhaimi Ibrahim, « Camouflage in malware: from encryption to metamorphism », *in*: *International Journal of Computer Science and Network Security* 12.*8* (2012), pp. 74–83.

[28] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis, « Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware », *in*: *Proceedings of the Seventh European Workshop on System Security*, 2014.

[29] Jinho Jung, Chanil Jeon, Max Wolotsky, Insu Yun, and Taesoo Kim, « AVPASS: Leaking and Bypassing Antivirus Detection Model Automatically », *in*: *Black Hat USA Briefings (Black Hat USA), Las Vegas, NV* (2017).

[30] Justin Sahs and Latifur Khan, « A machine learning approach to android malware detection », *in*: *2012 European Intelligence and Security Informatics Conference*, IEEE, 2012, pp. 141–147.

[31] Guillermo Suarez-Tangil, Santanu Kumar Dash, Mansour Ahmadi, Johannes Kinder, Giorgio Giacinto, and Lorenzo Cavallaro, « Droidsieve: Fast and accurate classification of obfuscated android malware », *in*: *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017, pp. 309–320.

[32] Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss, « "Andromaly": a behavioral malware detection framework for android devices », *in*: *Journal of Intelligent Information Systems* (2012).

[33] X. Chen, C. Li, D. Wang, S. Wen, J. Zhang, S. Nepal, Y. Xiang, and K. Ren, « Android HIV: A Study of Repackaging Malware for Evading Machine-Learning Detection », *in*: *IEEE Transactions on Information Forensics and Security* (2019), pp. 1–1.

[34] Hung Dang, Yue Huang, and Ee-Chien Chang, « Evading classifiers by morphing in the dark », *in*: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 119–133.

[35] Roberto Jordaney, Kumar Sharad, Santanu K. Dash, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro, « Transcend: Detecting Concept Drift in Malware Classification Models », *in*: 2017, pp. 625–642, URL: `https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/jordaney`.

[36] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, Akinori Ihara, and Kenichi Matsumoto, « The impact of mislabelling on the performance and

interpretation of defect prediction models », *in*: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, IEEE, 2015, pp. 812–823.

[37] Mahmoud Hammad, Joshua Garcia, and Sam Malek, « A Large-scale Empirical Study on the Effects of Code Obfuscations on Android Apps and Anti-malware Products », *in*: *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, ACM, 2018, pp. 421–431, URL: http://doi.acm.org/10.1145/3180155.3180228.

[38] Mourad Leslous, Val é rie Viet Triem Tong, Jean-Fran ç ois Lalande, and Thomas Genet, « GPFinder: Tracking the invisible in Android malware », *in*: *International Conference on Malicious and Unwanted Software (MALWARE)*, IEEE, 2017.

[39] « TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time », *in*: 2019, URL: https://www.usenix.org/conference/usenixsecurity19/presentation/pendlebury.

[40] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou, « Deep ground truth analysis of current android malware », *in*: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2017, pp. 252–276.

[41] J. Chen, C. Wang, Z. Zhao, K. Chen, R. Du, and G. Ahn, « Uncovering the Face of Android Ransomware: Characterization and Real-Time Detection », *in*: *IEEE Transactions on Information Forensics and Security* 13.5 (May 2018), pp. 1286–1300.

[42] *VirusTotal*, https://www.virustotal.com, (Accessed on 08/10/2020).

[43] Aleieldin Salem, Sebastian Banescu, and Alexander Pretschner, « Maat: Automatically Analyzing VirusTotal for Accurate Labeling and Effective Malware Detection », *in*: *arXiv e-prints* (2020), arXiv–2007.

[44] S. Rasthofer, S. Arzt, E. Lovat, and E. Bodden, « DroidForce: Enforcing Complex, Data-centric, System-wide Policies in Android », *in*: *2014 Ninth International Conference on Availability, Reliability and Security*, Sept. 2014, pp. 40–49.

[45] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber, « On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis », *in*: 2016, pp. 1101–1118,

URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/backes_android.

[46] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie, « Pscout: analyzing the android permission specification », *in*: *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 217–228.

[47] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang, « CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities », *in*: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, ACM, 2012, pp. 229–240, URL: http://doi.acm.org/10.1145/2382196.2382223.

[48] Mengtao Sun and Gang Tan, « NativeGuard: protecting android applications from third-party native libraries », *in*: ACM Press, 2014, pp. 165–176, URL: http://dl.acm.org/citation.cfm?doid=2627393.2627396.

[49] Joel Reardon, Á lvaro Feal, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, and Serge Egelman, « 50 ways to leak your data: An exploration of apps' circumvention of the android permissions system », *in*: *USENIX Security 19)*, 2019, pp. 603–620.

[50] Michael C. Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang, « Systematic Detection of Capability Leaks in Stock Android Smartphones », *in*: *NDSS*, 2012.

[51] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel, « FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps », *in*: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, ACM, 2014, pp. 259–269, URL: http://doi.acm.org/10.1145/2594291.2594299.

[52] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth, « TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones », *in*: *ACM Trans. Comput. Syst.* 32.*2* (June 2014), 5:1–5:29, URL: http://doi.acm.org/10.1145/2619091.

[53] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel, « IccTA: Detecting Inter-component Privacy Leaks in Android Apps », *in*: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, IEEE Press, 2015, pp. 280–291, ISBN: 978-1-4799-1934-5.

[54] *Exodus Privacy*, `https://exodus-privacy.eu.org/en/`, (Accessed on 08/02/2020).

[55] Mohannad Alhanahnah, Qiben Yan, Hamid Bagheri, Hao Zhou, Yutaka Tsutano, Witawas Srisa-an, and Xiapu Luo, « Detecting Vulnerable Android Inter-App Communication in Dynamically Loaded Code », *in*: *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, IEEE, Apr. 2019, pp. 550–558, URL: `https://ieeexplore.ieee.org/document/8737637/`.

[56] Youn Kyu Lee, Peera Yoodee, Arman Shahbazian, Daye Nam, and Nenad Medvidovic, « SEALANT: A Detection and Visualization Tool for Inter-app Security Vulnerabilities in Android », *in*: *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, IEEE Press, 2017, pp. 883–888, URL: `http://dl.acm.org/citation.cfm?id=3155562.3155672`.

[57] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon, « Effective Inter-component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis », *in*: *22nd USENIX Security Symposium - (USENIX Security 2013)*, SEC'13, USENIX Association, 2013, pp. 543–558, URL: `http://dl.acm.org/citation.cfm?id=2534766.2534813`.

[58] Roee Hay, Omer Tripp, and Marco Pistoia, « Dynamic detection of inter-application communication vulnerabilities in Android », *in*: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 118–128.

[59] *Intent | Android Developers*, `https://developer.android.com/reference/android/content/Intent`, (Accessed on 09/13/2020).

[60] Sven Bugiel, Stephen Heuser, and Ahmad-Reza Sadeghi, « Flexible and Fine-grained Mandatory Access Control on Android for Diverse Security and Privacy Policies », *in*: 2013, pp. 131–146, URL: `https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/bugiel`.

[61] Yiming Jing, Gail-Joon Ahn, Adam Doupé, and Jeong Hyun Yi, « Checking Intent-based Communication in Android with Intent Space Analysis », *in*: ACM Press, 2016, pp. 735–746, URL: http://dl.acm.org/citation.cfm?doid=2897845.2897904.

[62] Mu Zhang and Heng Yin, « AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications », *in*: *Proceedings 2014 Network and Distributed System Security Symposium*, Internet Society, 2014, URL: https://www.ndss-symposium.org/ndss2014/programme/appsealer-automatic-generation-vulnerability-specific-patches-preventing-component-hijacking/.

[63] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky, « Boxify: Full-fledged app sandboxing for stock android », *in*: *24th USENIX Security Symposium - (USENIX Security 2015)*, 2015, pp. 691–706.

[64] Antonio Bianchi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna, « NJAS: Sandboxing Unmodified Applications in Non-rooted Devices Running Stock Android », *in*: *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '15, ACM, 2015, pp. 27–38, URL: http://doi.acm.org/10.1145/2808117.2808122.

[65] Jing Yu and Toshihiro Yamauchi, « Access control to prevent attacks exploiting vulnerabilities of webview in android OS », *in*: *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, IEEE, 2013, pp. 1628–1633.

[66] Guliz Seray Tuncay, Soteris Demetriou, and Carl A Gunter, « Draco: A system for uniform and fine-grained access control for web code on android », *in*: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 104–115.

[67] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy, « Privilege escalation attacks on android », *in*: *international conference on Information security*, Springer, 2010, pp. 346–360.

[68] Luyi Xing, Xiaorui Pan, Rui Wang, Kan Yuan, and XiaoFeng Wang, « Upgrading your android, elevating my malware: Privilege escalation through mobile os updating », *in*: *2014 IEEE Symposium on Security and Privacy*, IEEE, 2014, pp. 393–408.

[69] Erika Chin and David Wagner, « Bifocals: Analyzing WebView Vulnerabilities in Android Applications », *in*: *Information Security Applications*, Springer, Cham, Aug. 2013, pp. 138–159, URL: https://link.springer.com/chapter/10.1007/978-3-319-05149-9_9.

[70] *Security-Enhanced Linux in Android Android Open Source Project*, https://source.android.com/security/selinux, (Accessed on 08/23/2020).

[71] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin, « Attacks on WebView in the Android system », *in*: *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011, pp. 343–352.

[72] AB Bhavani, « Cross-site scripting attacks on android webview », *in*: *arXiv preprint arXiv:1304.7451* (2013).

[73] *2011 Mobile Threats Report*, https://www.juniper.net/us/en/local/pdf/additional-resources/jnpr-2011-mobile-threats-report.pdf, (Accessed on 08/23/2020).

[74] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning, « Detecting repackaged smartphone applications in third-party android marketplaces », *in*: *Proceedings of the second ACM conference on Data and Application Security and Privacy*, 2012, pp. 317–326.

[75] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou, « Fast, scalable detection of" Piggybacked" mobile applications », *in*: *Proceedings of the third ACM conference on Data and application security and privacy*, 2013, pp. 185–196.

[76] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiao-yong Zhou, and XiaoFeng Wang, « Effective and efficient malware detection at the end host. », *in*: *USENIX security symposium*, vol. 4, 1, 2009, pp. 351–366.

[77] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning, « Detecting Repackaged Smartphone Applications in Third-party Android Marketplaces », *in*: *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, CODASPY '12, ACM, 2012, pp. 317–326, ISBN: 978-1-4503-1091-8, DOI: 10.1145/2133601.2133640.

[78] Keehyung Kim and Byung-Ro Moon, « Malware detection based on dependency graph using hybrid genetic algorithm », *in*: *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, 2010, pp. 1211–1218.

[79] Ilsun You and Kangbin Yim, « Malware obfuscation techniques: A brief survey », *in*: *2010 International conference on broadband, wireless computing, communication and applications*, IEEE, 2010, pp. 297–300.

[80] Naser Peiravian and Xingquan Zhu, « Machine learning for android malware detection using permission and api calls », *in*: *2013 IEEE 25th international conference on tools with artificial intelligence*, IEEE, 2013, pp. 300–305.

[81] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli, « MADAM: Effective and Efficient Behavior-based Android Malware Detection and Prevention », *in*: *IEEE Transactions on Dependable and Secure Computing* 15.*1* (Jan. 2018), pp. 83–97.

[82] Martina Lindorfer, Matthias Neugschwandtner, and Christian Platzer, « MARVIN: Efficient and Comprehensive Mobile App Classification through Static and Dynamic Analysis », *in*: *2015 IEEE 39th Annual Computer Software and Applications Conference*, IEEE, July 2015, pp. 422–433, URL: http://ieeexplore.ieee.org/document/7273650/.

[83] Andrea Saracino, Daniele Sgandurra, Gianluca Dini, and Fabio Martinelli, « Madam: Effective and efficient behavior-based android malware detection and prevention », *in*: *Transactions on Dependable and Secure Computing* (2016).

[84] Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli, « Adversarial malware binaries: Evading deep learning for malware detection in executables », *in*: *2018 26th European Signal Processing Conference (EUSIPCO)*, IEEE, 2018, pp. 533–537.

[85] Yuping Li, Jiyong Jang, Xin Hu, and Xinming Ou, « Android malware clustering through malicious payload mining », *in*: *International Symposium on Research in Attacks, Intrusions, and Defenses*, Springer, 2017, pp. 192–214.

[86] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto, « Stealth attacks: An extended insight into the obfuscation effects on android malware », *in*: *Computers & Security* 51 (2015), pp. 16–31.

[87] *Java Obfuscator and Android App Optimizer | ProGuard*, `https://www.guardsquare.com/en/products/proguard`, (Accessed on 09/13/2020).

[88] *Contagio dataset*, `http://contagiodump.blogspot.com/`, (Accessed on 09/12/2019).

[89] Darell JJ Tan, Tong-Wei Chua, and Vrizlynn LL Thing, « Securing android: a survey, taxonomy, and challenges », *in*: *ACM Computing Surveys (CSUR)* 47.*4* (2015), pp. 1–45.

[90] Blaine Nelson, Marco Barreno, Fuching Jack Chi, Anthony D Joseph, Benjamin IP Rubinstein, Udam Saini, Charles A Sutton, J Doug Tygar, and Kai Xia, « Exploiting Machine Learning to Subvert Your Spam Filter. », *in*: *LEET* 8 (2008), pp. 1–9.

[91] Benjamin IP Rubinstein, Blaine Nelson, Ling Huang, Anthony D Joseph, Shing-hon Lau, Satish Rao, Nina Taft, and J Doug Tygar, « Antidote: understanding and defending against poisoning of anomaly detectors », *in*: *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*, 2009, pp. 1–14.

[92] Battista Biggio, Blaine Nelson, and Pavel Laskov, « Poisoning attacks against support vector machines », *in*: *arXiv preprint arXiv:1206.6389* (2012).

[93] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Š rndi ć, Pavel Laskov, Giorgio Giacinto, and Fabio Roli, « Evasion attacks against machine learning at test time », *in*: *Joint European conference on machine learning and knowledge discovery in databases*, Springer, 2013, pp. 387–402.

[94] Ricardo N Rodrigues, Lee Luan Ling, and Venu Govindaraju, « Robustness of multimodal biometric fusion methods against spoof attacks », *in*: *Journal of Visual Languages & Computing* 20.*3* (2009), pp. 169–179.

[95] Huang Xiao, Battista Biggio, Gavin Brown, Giorgio Fumera, Claudia Eckert, and Fabio Roli, « Is feature selection secure against training data poisoning? », *in*: *International Conference on Machine Learning*, 2015, pp. 1689–1698.

[96] Juan Zheng, Zhimin He, and Zhe Lin, « Hybrid adversarial sample crafting for black-box evasion attack », *in*: *2017 International Conference on Wavelet Analysis and Pattern Recognition (ICWAPR)*, IEEE, 2017, pp. 236–242.

[97] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy, « Explaining and harnessing adversarial examples », *in*: *arXiv preprint arXiv:1412.6572* (2014).

[98] Yanpei Liu, Xinyun Chen, Chang Liu, and Dawn Song, « Delving into transferable adversarial examples and black-box attacks », *in*: *arXiv preprint arXiv:1611.02770* (2016).

[99] Gamaleldin Elsayed, Shreya Shankar, Brian Cheung, Nicolas Papernot, Alexey Kurakin, Ian Goodfellow, and Jascha Sohl-Dickstein, « Adversarial examples that fool both computer vision and time-limited humans », *in*: *Advances in Neural Information Processing Systems*, 2018, pp. 3910–3920.

[100] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel, « Adversarial Perturbations Against Deep Neural Networks for Malware Classification », *in*: *arXiv:1606.04435 [cs]* (June 2016), URL: `http://arxiv.org/abs/1606.04435`.

[101] Weiwei Hu and Ying Tan, « Generating Adversarial Malware Examples for Black-Box Attacks Based on GAN », *in*: *arXiv:1702.05983 [cs]* (Feb. 2017), URL: `http://arxiv.org/abs/1702.05983`.

[102] W Xu, Y Qi, and D Evans, *Automatically Evading Classifiers: A Case Study on PDF Malware Classifiers. NDSS*, 2016.

[103] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel, « Adversarial Examples for Malware Detection », *in*: *Computer Security – ESORICS 2017*, ed. by Simon N. Foley, Dieter Gollmann, and EinarEditors Snekkenes, Lecture Notes in Computer Science, Springer International Publishing, 2017, pp. 62–79.

[104] Yinxing Xue, Guozhu Meng, Yang Liu, Tian Huat Tan, Hongxu Chen, Jun Sun, and Jie Zhang, « Auditing Anti-Malware Tools by Evolving Android Malware and Dynamic Loading Technique », *in*: *IEEE Transactions on Information Forensics and Security* 12.7 (July 2017), pp. 1529–1544, URL: `http://ieeexplore.ieee.org/document/7837653/`.

[105] Khaled Bakour, Halil Murat Ünver, and Razan Ghanem, « A Deep Camouflage: Evaluating Android's Anti-malware Systems Robustness Against Hybridization of Obfuscation Techniques with Injection Attacks », *in*: *Arabian Journal for Science*

and Engineering (Aug. 2019), URL: http://link.springer.com/10.1007/s13369-019-04081-5.

[106] Wei Yang, Deguang Kong, Tao Xie, and Carl A. Gunter, « Malware Detection in Adversarial Settings: Exploiting Feature Evolutions and Confusions in Android Apps », *in*: *Proceedings of the 33rd Annual Computer Security Applications Conference*, ACSAC 2017, ACM, 2017, pp. 288–302, URL: http://doi.acm.org/10.1145/3134600.3134642.

[107] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang, « DroidChameleon: evaluating Android anti-malware against transformation attacks », *in*: *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security - ASIA CCS '13*, ACM Press, 2013, p. 329, URL: http://dl.acm.org/citation.cfm?doid=2484313.2484355.

[108] Keith Dillon, « Feature-level Malware Obfuscation in Deep Learning », *in*: *arXiv preprint arXiv:2002.05517* (2020).

[109] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro, « Intriguing Properties of Adversarial ML Attacks in the Problem Space », *in*: *arXiv preprint arXiv:1911.02142* (2019).

[110] Guozhu Meng, Yinxing Xue, Chandramohan Mahinthan, Annamalai Narayanan, Yang Liu, Jie Zhang, and Tieming Chen, « Mystique: Evolving Android Malware for Auditing Anti-Malware Tools », *in*: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, ACM, 2016, pp. 365–376, URL: http://doi.acm.org/10.1145/2897845.2897856.

[111] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck, « Structural detection of android malware using embedded call graphs », *in*: *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, 2013, pp. 45–54.

[112] Kevin Allix, Tegawendé François D Assise Bissyande, Jacques Klein, and Yves Le Traon, *Machine Learning-Based Malware Detection for Android Applications: History Matters!*, tech. rep., University of Luxembourg, SnT, 2014.

[113] Joshua Garcia, Mahmoud Hammad, Bahman Pedrood, Ali Bagheri-Khaligh, and Sam Malek, « Obfuscation-resilient, efficient, and accurate detection and family identification of android malware », *in*: *Department of Computer Science, George Mason University, Tech. Rep* 202 (2015).

[114] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster, « Scandroid: Automated security certification of android applications », *in*: *Manuscript, Univ. of Maryland, http://www. cs. umd. edu/avik/projects/scandroidascaa* 2.*3* (2009).

[115] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard, « Information flow analysis of android applications in droidsafe. », *in*: *NDSS*, vol. 15, 201, 2015, p. 110.

[116] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck, « Appcontext: Differentiating malicious and benign mobile app behaviors using context », *in*: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, IEEE, 2015, pp. 303–313.

[117] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna, « Execute this! analyzing unsafe and malicious dynamic code loading in android applications. », *in*: *NDSS*, vol. 14, 2014, pp. 23–26.

[118] Robert Podschwadt and Hassan Takabi, « On Effectiveness of Adversarial Examples and Defenses for Malware Classification », *in*: *International Conference on Security and Privacy in Communication Systems*, Springer, 2019, pp. 380–393.

[119] *Platform Architecture | Android Developers*, `https://developer.android.com/guide/platform`, (Accessed on 09/14/2020).

[120] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry, « Towards Taming Privilege-Escalation Attacks on Android. », *in*: *NDSS*, vol. 17, 2012, p. 19.

[121] Jinseong Jeon, Kristopher K Micinski, Jeffrey A Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S Foster, and Todd Millstein, « Dr. Android and Mr. Hide: fine-grained permissions in android applications », *in*: *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, 2012, pp. 3–14.

[122] Siegfried Rasthofer, Steven Arzt, Enrico Lovat, and Eric Bodden, « Droidforce: Enforcing complex, data-centric, system-wide policies in android », *in*: *International Conference on Availability, Reliability and Security*, 2014.

[123] Kevin Allix, Quentin Jerome, Tegawendé F. Bissyandé, Jacques Klein, Radu State, and Yves Le Traon, « A Forensic Analysis of Android Malware – How is Malware Written and How it Could Be Detected? », *in*: *2014 IEEE 38th Annual Computer Software and Applications Conference*, July 2014, pp. 384–393.

[124] *ViewOverlay | Android Developers*, `https://developer.android.com/reference/android/view/ViewOverlay`, (Accessed on 09/22/2020).

[125] Nick Benton, « Simple relational correctness proofs for static analyses and program transformations », *in*: *ACM SIGPLAN Notices* 39.*1* (2004), pp. 14–25.

[126] Heila van der Merwe, Brink van der Merwe, and Willem Visser, « Verifying android applications using Java PathFinder », *in*: *ACM SIGSOFT Software Engineering Notes* 37.*6* (2012), pp. 1–5.

[127] Eric Goubault and Sylvie Putot, « Robustness analysis of finite precision implementations », *in*: *Asian Symposium on Programming Languages and Systems*, Springer, 2013, pp. 50–57.

[128] Brian Chess and Gary McGraw, « Static analysis for security », *in*: *IEEE security & privacy* 2.*6* (2004), pp. 76–79.

[129] Guangdong Bai, Quanqi Ye, Yongzheng Wu, Heila Botha, Jun Sun, Yang Liu, Jin Song Dong, and Willem Visser, « Towards model checking android applications », *in*: *IEEE Transactions on Software Engineering* 44.*6* (2017), pp. 595–612.

[130] Abhijit Bose, Xin Hu, Kang G Shin, and Taejoon Park, « Behavioral detection of malware on mobile handsets », *in*: *Proceedings of the 6th international conference on Mobile systems, applications, and services*, 2008, pp. 225–238.

[131] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X. Sean Wang, and Binyu Zang, « Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis », *in*: *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, ACM, 2013, pp. 611–622, URL: `http://doi.acm.org/10.1145/2508859.2516689`.

[132] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, « TriggerScope: Towards Detecting Logic Bombs in Android Applications », *in*: *2016 IEEE Symposium on Security and Privacy (SP)*, May 2016, pp. 377–396.

[133] Michael Bierma, Eric Gustafson, Jeremy Erickson, David Fritz, and Yung Ryn Choe, « Andlantis: Large-scale Android dynamic analysis », *in*: *arXiv preprint arXiv:1410.7751* (2014).

[134] Lok Kwong Yan and Heng Yin, « DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis », *in*: 2012, pp. 569–584, URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/yan.

[135] Jonathan Crussell, Clint Gibler, and Hao Chen, « AnDarwin: Scalable Detection of Semantically Similar Android Applications », *in*: *Computer Security – ESORICS 2013*, ed. by Jason Crampton, Sushil Jajodia, and KeithEditors Mayes, Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 182–199.

[136] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, Xiaofeng Wang, and Kehuan Zhang, « Understanding Android Obfuscation Techniques: A Large-Scale Investigation in the Wild », *in*: *arXiv:1801.01633 [cs]* (Jan. 2018), URL: http://arxiv.org/abs/1801.01633.

[137] Z. Qu, S. Alam, Y. Chen, X. Zhou, W. Hong, and R. Riley, « DyDroid: Measuring Dynamic Code Loading and Its Security Implications in Android Applications », *in*: *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2017, pp. 415–426.

[138] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang, « RiskRanker: Scalable and Accurate Zero-day Android Malware Detection », *in*: *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, ACM, 2012, pp. 281–294, URL: http://doi.acm.org/10.1145/2307636.2307663.

[139] Jin-Young Kim, Seok-Jun Bu, and Sung-Bae Cho, « Zero-day malware detection using transferred generative adversarial networks based on deep autoencoders », *in*: *Information Sciences* 460–461 (Sept. 2018), pp. 83–102, URL: https://linkinghub.elsevier.com/retrieve/pii/S0020025518303475.

[140] *apklab.io*, https://www.apklab.io/, (Accessed on 08/23/2020).

[141] Matthew G Schultz, Eleazar Eskin, F Zadok, and Salvatore J Stolfo, « Data mining methods for detection of new malicious executables », *in*: *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*, IEEE, 2000, pp. 38–49.

[142] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz, « Automatic analysis of malware behavior using machine learning », *in*: *Journal of Computer Security* 19.*4* (2011), pp. 639–668.

[143] Ivan Firdausi, Alva Erwin, Anto Satriyo Nugroho, et al., « Analysis of machine learning techniques used in behavior-based malware detection », *in*: *2010 second international conference on advances in computing, control, and telecommunication technologies*, IEEE, 2010, pp. 201–203.

[144] Philipp Probst, Bernd Bischl, and Anne-Laure Boulesteix, « Tunability: Importance of hyperparameters of machine learning algorithms », *in*: *arXiv preprint arXiv:1802.09596* (2018).

[145] Marc Claesen and Bart De Moor, « Hyperparameter search in machine learning », *in*: *arXiv preprint arXiv:1502.02127* (2015).

[146] Olivier Chapelle, Vladimir Vapnik, Olivier Bousquet, and Sayan Mukherjee, « Choosing Multiple Parameters for Support Vector Machines », *in*: *arXiv preprint arXiv:1502.02127* (2002).

[147] James S. Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl, « Algorithms for Hyper-Parameter Optimization », *in*: *Advances in Neural Information Processing Systems (NIPS)*, 2011.

[148] Matthias Feurer, Katharina Eggensperger, Stefan Falkner, Marius Lindauer, and Frank Hutter, « Practical automated machine learning for the automl challenge 2018 », *in*: *International Workshop on Automatic Machine Learning at ICML*, 2018.

[149] Andrew Bedford, Sébastien Garvin, Josée Desharnais, Nadia Tawbi, Hana Ajakan, Frédéric Audet, and Bernard Lebel, « Andrana: Quick and Accurate Malware Detection for Android », *in*: *Foundations and Practice of Security*, 2017.

[150] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar, « Hyperband: A novel bandit-based approach to hyperparameter optimization », *in*: *The Journal of Machine Learning Research* (2017).

[151] Shih-Wei Lin, Kuo-Ching Ying, Shih-Chieh Chen, and Zne-Jung Lee, « Particle swarm optimization for parameter determination and feature selection of support vector machines », *in*: *Expert systems with applications* (2008).

[152] Jinn-Tsong Tsai, Jyh-Horng Chou, and Tung-Kuan Liu, « Tuning the structure and parameters of a neural network by using hybrid Taguchi-genetic algorithm », *in*: *Transactions on Neural Networks* (2006).

[153] James Bergstra and Yoshua Bengio, « Random search for hyper-parameter optimization », *in*: *Journal of machine learning research* (2012).

[154] Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al., « Scikit-learn: Machine learning in Python », *in*: *Journal of machine learning research* (2011).

[155] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al., « Tensorflow: A system for large-scale machine learning », *in*: *Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[156] Edward G Coffman Jr, Michael R Garey, and David S Johnson, « An application of bin-packing to multiprocessor scheduling », *in*: *SIAM Journal on Computing* 7.*1* (1978), pp. 1–17.

[157] *Nomad by HashiCorp*, `https://www.nomadproject.io/`, (Accessed on 02/10/2020).

[158] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon, « AndroZoo: collecting millions of Android apps for the research community », *in*: *Working Conference on Mining Software Repositories (MSR)*, 2016.

[159] Brad Miller, Alex Kantchelian, Michael Carl Tschantz, Sadia Afroz, Rekha Bachwani, Riyaz Faizullabhoy, Ling Huang, Vaishaal Shankar, Tony Wu, George Yiu, and et al., « Reviewer Integration and Performance Measurement for Malware Detection », *in*: 2016.

[160] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon, « Are your training datasets yet relevant? », *in*: *International Symposium on Engineering Secure Software and Systems*, 2015.

[161] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro, « TESSERACT: Eliminating experimental bias in malware classification across space and time », *in*: *USENIX Security Symposium*, 2019.

[162] Anshul Arora and Sateesh K. Peddoju, « Minimizing Network Traffic Features for Android Mobile Malware Detection », *in*: *Proceedings of the 18th International Conference on Distributed Computing and Networking - ICDCN '17*, ACM Press, 2017, pp. 1–10, URL: http://dl.acm.org/citation.cfm?doid=3007748.3007763.

[163] Shuaifu Dai, Alok Tongaonkar, Xiaoyin Wang, Antonio Nucci, and Dawn Song, « Networkprofiler: Towards automatic fingerprinting of android apps », *in*: *2013 Proceedings IEEE INFOCOM*, IEEE, 2013, pp. 809–817.

[164] TaeGuen Kim, BooJoong Kang, Mina Rho, Sakir Sezer, and Eul Gyu Im, « A multimodal deep learning method for android malware detection using various features », *in*: *IEEE Transactions on Information Forensics and Security* 14.*3* (2018), pp. 773–788.

[165] Bhaskar Pratim Sarma, Ninghui Li, Chris Gates, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy, « Android permissions: a perspective combining risks and benefits », *in*: *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, 2012, pp. 13–22.

[166] Paolo Calciati and Alessandra Gorla, « How do apps evolve in their permission requests? a preliminary study », *in*: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, IEEE, 2017, pp. 37–41.

[167] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan, « Soot - a Java Bytecode Optimization Framework », *in*: *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, IBM Press, 1999, pp. 13–, URL: http://dl.acm.org/citation.cfm?id=781995.782008.

[168] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro, « Intriguing Properties of Adversarial ML Attacks in the Problem Space », *in*: *IEEE Symposium on Security and Privacy*, IEEE Computer Society, 2020, pp. 1308–1325, DOI: 10.1109/SP40000.2020.00073, URL: https://doi.ieeecomputersociety.org/10.1109/SP40000.2020.00073.

[169] Steven Arzt, Siegfried Rasthofer, and Eric Bodden, « Susi: A tool for the fully automated classification and categorization of android sources and sinks », *in*: *University of Darmstadt, Tech. Rep. TUDCS-2013* 114 (2013), p. 108.

[170] *Android ABIs | Android NDK | Android Developers*, https://developer.android.com/ndk/guides/abis, (Accessed on 08/29/2020).

[171] Sean Park, Iqbal Gondal, Joarder Kamruzzaman, and Jon Oliver, « Generative malware outbreak detection », *in*: *IEEE International Conference on Industry Technology ICIT, Melbourne*, 2019.

[172] *iBotPeaches/Apktool: A tool for reverse engineering Android apk files*, https://github.com/iBotPeaches/Apktool, (Accessed on 08/24/2020).

[173] *pxb1988/dex2jar: Tools to work with android .dex and java .class files*, https://github.com/pxb1988/dex2jar, (Accessed on 08/24/2020).

[174] *Most Android Apps can easily be decompiled to remove the ads*, https://blog.mmccoo.com/2017/02/28/most-android-apps-can/easily-be-decompiled-to-remove-the-ads/, (Accessed on 05/27/2019).

[175] Siqi Ma, David Lo, Teng Li, and Robert H. Deng, « CDRep: Automatic Repair of Cryptographic Misuses in Android Applications », *in*: ACM, May 2016, pp. 711–722, ISBN: 978-1-4503-4233-9, DOI: 10.1145/2897845.2897896.

[176] Ding Li and William G. J. Halfond, « Optimizing energy of HTTP requests in Android applications », *in*: ACM, Aug. 2015, pp. 25–28, ISBN: 978-1-4503-3815-8, DOI: 10.1145/2804345.2804351.

[177] Lavoisier Wapet, Alain Tchana, Giang Son Tran, and Daniel Hagimont, « Preventing the propagation of a new kind of illegitimate apps », *in*: *Future Generation Computer Systems* 94 (May 2019), pp. 368–380, ISSN: 0167-739X, DOI: 10.1016/j.future.2018.11.051.

[178] Jin-Hyuk Jung, Ju Young Kim, Hyeong-Chan Lee, and Jeong Hyun Yi, « Repackaging Attack on Android Banking Applications and Its Countermeasures », *in*: *Wireless Personal Communications* 73.*4* (Dec. 2013), pp. 1421–1437, ISSN: 1572-834X, DOI: 10.1007/s11277-013-1258-x.

[179] Tony Bradley, *DroidDream Becomes Android Market Nightmare | PCWorld*, https://www.pcworld.com/article/221247/droiddream_becomes_android_market_nightmare.html, (Accessed on 11/11/2019).

[180] *woxihuannisja/Bangcle: The second generation Android Hardening Protection*, https://github.com/woxihuannisja/Bangcle, (Accessed on 08/23/2020).

[181] Yueqian Zhang, Xiapu Luo, and Haoyang Yin, « Dexhunter: toward extracting hidden code from packed android applications », *in*: *European Symposium on Research in Computer Security*, Springer, 2015, pp. 293–311.

[182] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Junliang Shu, Bodong Li, Wenjun Hu, and Dawu Gu, « Appspear: Bytecode decrypting and dex reassembling for packed android malware », *in*: *International Symposium on Recent Advances in Intrusion Detection*, Springer, 2015, pp. 359–381.

[183] Erik Ramsgaard Wognsen and Henrik S ø ndberg Karlsen, « Static analysis of Dalvik bytecode and reflection in Android », *in*: *Master's thesis, Department of Computer Science, Aalborg University, Aalborg, Denmark* (2012).

[184] *Android Developers Blog: Backward compatibility for Android applications*, `https://android-developers.googleblog.com/2009/04/backward-compatibility-for-android.html`, (Accessed on 05/27/2019).

[185] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, Xiaofeng Wang, and Kehuan Zhang, « Understanding Android obfuscation techniques: A large-scale investigation in the wild », *in*: *International Conference on Security and Privacy in Communication Systems*, Springer, 2018, pp. 172–192.

[186] P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. dAmorim, and M. D. Ernst, « Static Analysis of Implicit Control Flow: Resolving Java Reflection and Android Intents (T) », *in*: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2015, pp. 669–679, DOI: `10.1109/ASE.2015.69`.

[187] *Security tips | Android Developers*, `https://developer.android.com/training/articles/security-tips`, (Accessed on 05/27/2019).

[188] *ijiami/ApkProtect*, `https://github.com/ijiami/ApkProtect`, (Accessed on 08/23/2020).

[189] *About Qihoo 360 | 360 Total Security*, `https://www.360totalsecurity.com/en/about/`, (Accessed on 08/23/2020).

[190] Z. Qu, S. Alam, Y. Chen, X. Zhou, W. Hong, and R. Riley, « DyDroid: Measuring Dynamic Code Loading and Its Security Implications in Android Applications », *in*: *International Conference on Dependable Systems and Networks (DSN)*, June 2017, DOI: `10.1109/DSN.2017.14`.

[191] Yury Zhauniarovich, Maqsood Ahmad, Olga Gadyatskaya, Bruno Crispo, and Fabio Massacci, « StaDynA: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications », *in*: *Conference on Data and Application Security and Privacy (CODASPY)*, event-place: San Antonio, Texas, USA, ACM, 2015, ISBN: 978-1-4503-3191-3, DOI: 10.1145/2699026.2699105.

[192] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang, « Catch me if you can: Evaluating android anti-malware against transformation attacks », *in*: *IEEE Transactions on Information Forensics and Security* 9.*1* (2013), pp. 99–108.

[193] Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Manoj Singh Gaur, Mauro Conti, and Muttukrishnan Rajarajan, « Evaluation of android anti-malware techniques against dalvik bytecode obfuscation », *in*: *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*, IEEE, 2014, pp. 414–421.

[194] Li Li, Tegawendé F. Bissyandé Li Daoyuan, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro, « Understanding Android App Piggybacking: A Systematic Study of Malicious Code Grafting », *in*: *Transactions on Information Forensics and Security (TIFS)* (2017).

[195] Yue Duan, Mu Zhang, Abhishek Vasisht Bhaskar, Heng Yin, Xiaorui Pan, Tongxin Li, Xueqiang Wang, and XiaoFeng Wang, « Things You May Not Know About Android (Un)Packers: A Systematic Study based on Whole-System Emulation », *in*: *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, The Internet Society, 2018.

[196] Andreas Moser, Christopher Kruegel, and Engin Kirda, « Limits of static analysis for malware detection », *in*: *Annual Computer Security Applications Conference (ACSAC)*, IEEE, 2007.

[197] M. Sun, X. Li, J. C. S. Lui, R. T. B. Ma, and Z. Liang, « Monet: A User-Oriented Behavior-Based Malware Variants Detection System for Android », *in*: *Transactions on Information Forensics and Security (TIFS)* (2017).

[198] Nitesh V. Chawla, Nathalie Japkowicz, and Aleksander Kotcz, « Editorial: Special Issue on Learning from Imbalanced Data Sets », *in*: *ACM SIGKDD explorations newsletter* (2004).

[199] Qiang Yang and Xindong Wu, « 10 challenging problems in data mining research », *in*: *International Journal of Information Technology 38; Decision Making* (2006).

[200] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken, « Apposcopy: Semantics-based detection of android malware through static analysis », *in*: *International Symposium on Foundations of Software Engineering (FSE)*, 2014.

[201] *Google Acquires Online Virus, Malware and URL Scanner VirusTotal*, `http://tcrn.ch/P20EHo`, TechChrunch.

[202] Siegfried Rasthofer, Steven Arzt, and Eric Bodden, « A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. », *in*: *Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

[203] Min Zheng, Patrick P. C. Lee, and John C. S. Lui, « ADAM: An Automatic and Extensible Platform to Stress Test Android Anti-virus Systems », *in*: *Detection of Intrusions and Malware, and Vulnerability Assessment*, ed. by Ulrich Flegel, Evangelos Markatos, and WilliamEditors Robertson, Springer, 2013, ISBN: 978-3-642-37300-8.

[204] Dimitris Bertsimas, John Tsitsiklis, et al., « Simulated annealing », *in*: *Statistical science* (1993).

[205] *Kaspersky Antivirus 2020 pour Android | Kaspersky*, `https://www.kaspersky.fr/android-security`, (Accessed on 09/23/2020).

[206] *Statistics - VirusTotal*, `https://www.virustotal.com/en/statistics/`, (Accessed on 08/25/2020).

[207] *How it works – VirusTotal*, `https://support.virustotal.com/hc/en-us/articles/115002126889-How-it-works`, (Accessed on 08/10/2020).

[208] Zhou Wang, Eero P Simoncelli, and Alan C Bovik, « Multiscale structural similarity for image quality assessment », *in*: *The Thrity-Seventh Asilomar Conference on Signals, Systems & Computers, 2003*, vol. 2, Ieee, 2003, pp. 1398–1402.

[209] Stuart Millar, Niall McLaughlin, Jesus Martinez del Rincon, Paul Miller, and Ziming Zhao, « DANdroid: A Multi-View Discriminative Adversarial Network for Obfuscated Android Malware Detection », *in*: *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*, CODASPY '20, Association for Computing Machinery, Mar. 2020, pp. 353–364, URL: `https://doi.org/10.1145/3374664.3375746`.

[210] Siegfried Rasthofer, Irfan Asrar, Stephan Huber, and Eric Bodden, « How current android malware seeks to evade automated code analysis », *in*: *IFIP International Conference on Information Security Theory and Practice*, Springer, 2015, pp. 187–202.

[211] Suleiman Y Yerima and Sakir Sezer, « DroidFusion: a novel multilevel classifier fusion approach for android malware detection », *in*: *IEEE transactions on cybernetics* 49.*2* (2018), pp. 453–466.

[212] Suleiman Y Yerima, Sakir Sezer, and Igor Muttik, « Android malware detection using parallel machine learning classifiers », *in*: *2014 Eighth International Conference on Next Generation Mobile Apps, Services and Technologies*, IEEE, 2014, pp. 37–42.

[213] Jack W Stokes, John C Platt, Jonathan M Keller, Joseph L Faulhaber, Anil Francis Thomas, Adrian M Marinescu, Marius G Gheorghescu, and George Chicioreanu, *Malware detection using multiple classifiers*, US Patent App. 12/358,246, July 2010.

[214] Saurabh Panjwani and Achintya Prakash, « Crowdsourcing attacks on biometric systems », *in*: *10th Symposium On Usable Privacy and Security (SOUPS 2014)*, 2014, pp. 257–269.

[215] Gang Wang, Tianyi Wang, Haitao Zheng, and Ben Y Zhao, « Man vs. machine: Practical adversarial detection of malicious crowdsourcing workers », *in*: *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 239–254.

[216] Jon Oberheide and Charlie Miller, « Dissecting the android bouncer », *in*: *SummerCon2012, New York* 95 (2012), p. 110.

[217] *Google will now let you play Stadia on any modern Android phone - The Verge*, `https://www.theverge.com/2020/6/11/21288175/google-stadia-update-android-phone-support-cloud-gaming-mobile`, (Accessed on 09/28/2020).

[218] Yérom-David Bromberg and Louison Gitzinger, « DroidAutoML: A Microservice Architecture to Automate the Evaluation of Android Machine Learning Detection Systems », *in*: *IFIP International Conference on Distributed Applications and Interoperable Systems*, Springer, 2020, pp. 148–165.

Acknowledgement

My first thanks go to my parents, my sister and my brothers for finding the words to support me throughout these three years, and having all been here when I needed to complain.

A grateful thank to my love, Nine, who stayed by my side at all times, with an unfailing loyalty. She gave me the comfort I needed and found the patience to wait and endure my infinite deadlines, I cannot be more grateful.

Thank to Dorothée, Nicolas, Hugo, Clara and their partners, who all contributed to help me move forward in this work in their own distinctive way. I would specifically like to thank Charles, my true friend who showed me how cool are computer sciences and helped me a lot throughout my academic journey.

My two roommates, Lucien and Lola, also deserve their praise for taking care of me as their own child, for giving me courage every single day, for making me laugh in difficult times, without asking for anything in return.

Thanks to all my friends, who believed in me since the beginning of this work.

Thanks to all the WIDE's team, who made everyday life pleasant throughout the three years spent with the team. Quentin, Adrien, Alex, Loïck, Simon, for these moments of sharing and these frenzied discussions at lunch. I would specifically thank Quentin, my office colleague, who took a lot of time to teach me his knowledge, as well as his willing to make the world a better place. A cheerful thank to Virginie Desroches, our administration expert many would dream of, who helped me navigate the intricacies of administrative tasks in a joyful and relaxed atmosphere.

I would further like to thank my PhD advisor, David Bromberg, who trusted me after graduating from my engineering school and meticulously guided me throughout my thesis. I am grateful to him for his dedication to me and for offering me so many professional opportunities. Finally, many thanks to all the jury members who accepted to evaluate my work, especially Sonia Ben Mokthar and Romain Rouvoy for reporting on it.

UNIVERSITÉ DE RENNES 1

Titre : Survivre à la prolifération massive des malwares sur mobile

Mot clés : détection de malwares, périphériques intelligents, apprentissage statistique, Android, Attaques adversariales

Résumé : De nos jours, nous sommes entourés de périphériques intelligents autonomes qui interagissent avec de nombreux services dans le but d'améliorer notre niveau de vie. Ces périphériques font partie d'écosystèmes plus larges, dans lesquels de nombreuses entreprises collaborent pour faciliter la distribution d'applications entre les développeurs et les utilisateurs. Cependant, des personnes malveillantes en profitent illégalement pour infecter les appareils des utilisateurs avec des application malveillantes. Malgré tous les efforts mis en oeuvre pour défendre ces écosystèmes, le taux de périphériques infectés par des malware est toujours en augmentation en 2020. Dans cette thèse, nous explorons trois axes de recherche dans le but d'améliorer globalement la détection de malwares dans l'écosystème Android. Nous démontrons d'abord que la précision des systèmes de détection basés sur le machine learning peuvent être améliorés en automatisant leur évaluation et en ré-utilisant le concept d'*AutoML* pour affiner les paramètres des algorithmes d'apprentissage. Nous proposons une approche pour créer automatiquement des variantes de malwares à partir de combinaisons de techniques d'évasion complexes pour diversifier les datasets de malwares expérimentaux dans le but de mettre à l'épreuve les systèmes de détection. Enfin, nous proposons des méthodes pour améliorer la qualité des datasets expérimentaux utilisés pour entrainer et tester les systèmes de détection.

Title: Surviving the massive proliferation of mobile malware

Keywords: malware detection, smart devices, machine learning, Android, Adversarial attacks

Abstract: Nowadays, many of us are surrounded by smart devices that seamlessly operate interactively and autonomously together with multiple services to make our lives more confortable. These smart devices are part of larger ecosystems, in which various companies collaborate to ease the distribution of applications between developers and users. However malicious attackers take advantage of them illegitimately to infect users' smart devices with malicious applications. Despite all the efforts made to defend these ecosystems, the rate of devices infected with malware is still increasing in 2020. In this thesis, we explore three research axes with the aim of globally improving malware detection in the Android ecosystem. We demonstrate that the accuracy of machine learning-based detection systems can be improved by automating their evaluation and by reusing the concept of *AutoML* to fine-tune learning algorithms parameters. We propose an approach to automatically create malware variants from combinations of complex evasion techniques to diversify experimental malware datasets in order to challenge existing detection systems. Finally, we propose methods to globally increase the quality of experimental datasets used to train and test detection systems.