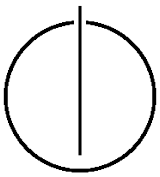
FAKULTÄT FÜR INFORMATIK

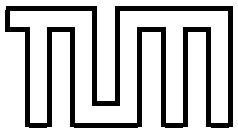
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Dissertation in Informatik

Accurate and Reliable Labeling for Effective Detection of Android Malware

Aleieldin Salem





FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Lehrstuhl IV - Software and Systems Engineering

Accurate and Reliable Labeling for Effective Detection of Android Malware

Aleieldin Salem

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Prof. Jens Großklags, Ph.D.
Prüfer der Dissertation:	
1.	Prof. Dr. Alexander Pretschner
2.	Prof. Lorenzo Cavallaro, King's College London, UK

Die Dissertation wurde am 05.10.2020 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 17.02.2021 angenommen.

Acknowledgments

There are no short cuts to any place worth going. لا يوجد طريق مختصر إلى مكان يستحق الذهاب إليه.

Amen! In accomplishing this feat, I am infinitely indebted to the following people.

My genuine gratitude goes to Prof. Dr. Alexander Pretschner for giving me the chance to work on what I am interested in and for his continuous support and encouragement. Doing this job, I learned how to be more observant and critical yet constructive. I had the opportunity to pass on whatever little knowledge I accumulated, and I learned how to tune out all distractions and focus on my goals. Thank you for the opportunity.

I would also like to thank my second supervisor, Prof. Lorenzo Cavallaro, and his group, particularly Dr. Fabio Pierazzi and Feargus Pendlebury, for their unparalleled hospitality, for the fruitful discussions that we had at King's, and for their technical help with my work. The enthusiasm and positive energy with which you guys pursue solutions for the problems our community faces is admirable. Kudos!

My colleagues and friends at the chair of Software and Systems Engineering designated the *Eyes-Prizecists*: our CB's, CLGBT's, Spinning, Grave-yarding, Suçuk Halloumi's, and inside jokes carried me through tough times. It is an honor and a privilege to know you and to have worked with you. Dr. Sebastian Banescu, thank you for the hours of discussions that helped to shape this thesis and its storyline. Amjad and Patrick, thank you for taking the time to read this thesis and for helping me to make it better. The young men and women that I had the privilege to teach or advise, thank you for your efforts and for helping me throughout my employment at the TUM.

My Egyptian friends, whether we went to school together, did our bachelor's together, served in the same unit, shared a cubicle in a company, or just randomly met in the streets of Munich, thank you for the encouraging messages, for visiting as often as you can, and for keeping me up-to-date with the ridicule and non-sense that manifests everyday in the old country.

To my family, you have always been my backbone. Thank you for believing in and supporting my scientific and academic endeavors, both emotionally and financially, without pressuring me to settle down somewhere, find a wife, and give you loads of grandchildren.

My German family, Dr. Marianne Blank-Huber and Theresa Huber, your love, care, and awesome food compensated for the absence of my biological family; thank you.

Above all, my incomparably better half, Katharina Huber, every time I utter the words "*Why the heck am I doing this?*", I remind myself that had it not been for this job, we would not have crossed paths. The best thing I got out of this is not a doctoral title, it is indeed you. Having said that, I guess credit goes to Saahil and Barbara for organizing that house-warming party in which we met. :)

Zusammenfassung

Seit Jahrzehnten wird an der Erkennung von Malware mit Hilfe maschinellen Lernens geforscht. Nichts desto trotz begegnet man schadhaften Anwendungen nach wie vor. Dies deutet darauf hin, dass die derzeit angewandten Methoden zur Identifizierung von Malware unzulänglich sind. Eines der Probleme der Methoden zur Identifizierung von Malware, das für diesen Mangel verantwortlich ist, ist das Problem, der genauen und zuverlässigen Kennzeichnung der Anwendungen, worauf die Erkennungsmethoden sodann aufbauen. Verwendet man Kennzeichnungen, die die Schadhaftigkeit von Anwendungen (im Folgenden Apps genannt) fehlerhaft wiedergeben, riskiert man, dass die Verlässlichkeit von Studien unterminiert wird, die die von solchen Apps verfolgten Trends untersuchen und, noch viel wichtiger, dass die Entwicklung wirksamer Methoden zur Identifizierung von Malware beeinträchtigt wird.

Da es sich letztlich nicht durchführen lässt, eine große Anzahl von Apps manuell zu analysieren und zu kennzeichnen, um exakte Labels zu erhalten, sind Wissenschaftler dazu gezwungen, sich auf Onlineplattformen, wie VirusTotal zu verlassen, die die Scan-Ergebnisse unterschiedlicher kommerzieller Anti-Virus-Software bereitstellen. Allerdings gibt es kein Standardverfahren zur Interpretation dieser Scan-Ergebnisse, um die Anwendungen zu kennzeichnen. Infolgedessen verlassen sich Wissenschaftler auf ihre Intuition und wenden Strategien an, die aus dem Stehgreif einen Grenzwert festlegen, um die Apps in den Datensätzen zu kennzeichnen, die ihre Erkennungsmethoden trainieren und bewerten, oder die sie den anderen Forschern als Maßstab zugänglich machen. Auch wenn einige der verwendeten Grenzwerte Apps genauer kennzeichnen als andere, hat die Dynamik von VirusTotal zur Folge, dass die Genauigkeit einst optimaler Grenzwerte abnehmen kann und die Nachhaltigkeit schwellenwertbasierter Strategien letztlich untergraben könnte. Obschon der Forschungsgemeinschaft bekannt, werden die Aspekte einer solchen Dynamik, wie sie sich in den Scan-Berichten von VirusTotal ausdrücken und wie diese grenzwertbasierte Kennzeichnungsstrategien beeinflussen, nicht explizit diskutiert. Vor diesem Hintergrund ist das Hauptanliegen dieser Arbeit anderen Wissenschaftlern Methoden zur Verfügung zu stellen, mit denen sie die Scan-Berichte von VirusTotal optimal nutzen können, bis eine stabilere und zuverlässigere Plattform zur Verfügung steht. Zu diesem Zweck werden die Aspekte der Dynamik von VirusTotal und wie diese die Leistung von grenzwertbasierten Kennzeichnungsstrategien beeinflusst, die über einen längeren Zeitraum feste Grenzwerte verwenden, aufgezeigt. Darauf aufbauend wird ein Algorithmus vorgeschlagen, der die gegenwärtig optimalen Grenzwerte zur Verwendung des Kennzeichnens von Android-Apps zu jedem beliebigen Zeitpunkt identifiziert.

Um es Wissenschaftlern und Wissenschaftlerinnen zu ersparen VirusTotal-Scan-Berichte jedes Mal, wenn sie eine Android-App mit Hilfe eines grenzwertbasierten Ansatzes klassifizieren wollen, aufs neue herunterzuladen und zu scannen, wird Maat implementiert. Maat ist eine systematische und automatisierte Methode, die VirusTotal Scan-Berichte von Apps analysiert und auf maschinellem Lernen basierende Erkennungsmethoden

trainiert, von denen sich gezeigt hat, dass sie die Kennzeichnungsgenauigkeit von hypothetisch grenzwertbasierten Kennzeichnungsstrategien, die über die Zeit hinweg immer den bestmöglichen Grenzwert verwenden, erreichen. Ferner tragen solche Erkennungsmethoden, die auf maschinellem Lernen basieren, dazu bei, effektivere Erkennungsmethoden zu trainieren, als ihre grenzwertbasierten Gegenstücke. Im Ergebnis stellen Maats Erkennungsmethoden, die auf maschinellem Lernen basieren eine brauchbare Alternative zu konventionellen grenzwertbasierten Erkennungsmethoden, die für die Dynamik von VirusTotal anfällig sind, dar. Zu guter Letzt werden die Grenzen von VirusTotal, die während dieser Forschung begegnet sind benannt und eine Architektur für eine stabilere und verlässlichere Plattform vorgeschlagen, die diese Begrenzungen umgeht.

Abstract

The problem of Machine Learning (ML)-based malware detection has been researched for decades. However, malicious applications continue to be found in the wild, which suggests the inadequacy of currently-adopted malware detection methods. Among the problems facing malware detection methods that might lead to such inadequacy is that of assigning accurate and reliable labeld to applications used to train and evaluate such methods. Using labels that inaccurately represent the malignancy of applications (hereafter apps) risks undermining the reliability of studies that inspect trends adopted by malicious apps and, more importantly, might impede the development of effective detection methods.

Manually analyzing and labeling large numbers of apps to get accurate labels is infeasible, which forces researchers to rely on online platforms, such as `VirusTotal`, that provide scan results from different commercial antiviral software. Unfortunately, there are no standard procedures for interpreting these scan results to label apps. Hence, researchers use their intuitions and adopt ad hoc threshold-based strategies to label the apps in the datasets used to train and evaluate their detection methods or release to the research community as benchmarks. Although some of the adopted thresholds may accurately label apps better than others, the dynamicity of `VirusTotal` means that the accuracy of once-optimal thresholds might depreciate, effectively undermining the sustainability of threshold-based strategies. Albeit known to the research community that `VirusTotal` is dynamic, the aspects of such dynamicity, how it manifests in `VirusTotal` scan reports, and how it impacts threshold-based labeling strategies are not clearly discussed.

In this context, the main objective of this thesis is to provide the research community with methods to optimally utilize `VirusTotal` scan reports until a more stable, reliable platform is implemented. To that end, we reveal the aspects of `VirusTotal`'s dynamicity and how it impacts the performance of threshold-based labeling strategies that use fixed thresholds over prolonged periods. Based on these findings, we propose an algorithm that identifies the currently optimal thresholds to use upon labeling Android apps at any point in time. To relieve researchers of the burden of re-scanning and downloading `VirusTotal` scan reports each time they wish to label Android apps using threshold-based labeling strategies, we implemented Maat, a systematic and automated method that analyzes `VirusTotal` scan reports of apps and trains ML-based detection methods that we found to mimic the labeling accuracies of hypothetical threshold-based labeling strategies that always utilize the best possible thresholds over time. Furthermore, such ML-based detection methods contribute to training more effective detection methods than their threshold-based counterparts. Effectively, Maat's ML-based detection methods provide a viable alternative to conventional threshold-based labeling strategies that are susceptible to `VirusTotal`'s dynamicity. Lastly, we enumerate all the limitations of `VirusTotal` we encountered throughout this research and propose an architecture of a more stable and reliable platform that mitigates those limitations.

Outline of the Thesis

CHAPTER 1: INTRODUCTION

This chapter introduces the problem tackled by this thesis and highlights its significance. It presents the objectives of this work, the research questions it poses, and the solution proposed to achieve such objectives and address those questions.

CHAPTER 2: ANDROID MALWARE

This chapter presents fundamental concepts about Android apps and the process of implementing malicious and potentially unwanted versions of such apps. The chapter then presents an overview of the trends and functionalities adopted by Android malware found in practice including the nature of the payloads found in them. Lastly, the chapter presents the subjectivity of labeling malicious apps belonging to different malware types. Parts of this chapter have previously appeared in peer-reviewed publications [8], [118], and [119] co-authored by the author of this thesis.

CHAPTER 3: ANDROID MALWARE DETECTION

This chapter provides an overview of Android malware detection, its theoretical limitations, and the objectives it pursues in practice. In this chapter, we focus on ML-based malware detection methods and enumerate the challenges hindering effective detection. Parts of this chapter have previously appeared in peer-reviewed publications [120] and [119], co-authored by the author of this thesis.

CHAPTER 4: THRESHOLD-BASED LABELING STRATEGIES

This chapter focuses on threshold-based labeling strategies, their structure, advantages, and disadvantages. It reveals their sensitivity to the dynamicity of `VirusTotal`, unveils some limitations of the platform, and how to work around them to choose more stable thresholds. Parts of this chapter have previously appeared in peer-reviewed publications [9], and [116], co-authored by the author of this thesis.

CHAPTER 5: MAAT: A FRAMEWORK TO OPTIMALLY UTILIZE VIRUSTOTAL

This chapter presents the main contribution of this thesis: a systematic method, called Maat, that automatically analyzes `VirusTotal` scan reports of pre-labeled apps to identify the set of correct and stable `VirusTotal` scanners and uses such information to build ML-based labeling strategies. We describe how the set of correct and stable scanners is identified along with insights about `VirusTotal` and its scanners that we made in this process. Parts of this chapter have previously appeared in peer-reviewed publications [9] and [116], co-authored by the author of this thesis.

CHAPTER 6: EVALUATING MAAT

This chapter evaluates Maat's ability to assign labels to apps based on their `VirusTotal` scan reports that accurately reflect their ground truths. The chapter also discusses the impact of such accurate labeling on the performance of ML-based detection methods and, in general, the role of accurate labeling on the performance of detection methods. Parts of this chapter have previously appeared in peer-reviewed publications [9] and [116], co-authored by the author of this thesis.

CHAPTER 7: AN ALTERNATIVE TO VIRUSTOTAL

This chapter recaps the limitations of `VirusTotal` that were unveiled throughout the thesis and details the design of an alternative platform that tackles the issues of `VirusTotal`, effectively providing the research community with a more reliable, consistent, and stable source of labels for Android apps. Parts of this chapter have previously appeared in peer-reviewed publication [116], co-authored by the author of this thesis.

CHAPTER 8: RELATED WORK

This chapter enumerates and discusses related work particularly in the fields of designing and evaluating Android malware analysis and detection methods and identifies research gaps relevant for this thesis. Parts of this chapter have previously appeared in peer-reviewed publications [9], [116], [8], [118], and [119] co-authored by the author of this thesis.

CHAPTER 9: CONCLUSIONS

This chapter summarizes the findings of this thesis and draws conclusions from them to address the research questions posed in Chapter 1. It discusses the limitations of this work and suggests different aspects on how to further enhance it.

N.B.: *Multiple chapters of this dissertation are based on different publications authored or co-authored by the author of this dissertation. Such publications are mentioned in the short descriptions above. Due to the obvious content overlapping, quotes from such publications within the respective chapters are not marked or cited explicitly. However, in Appendix H, we detail the parts taken from each paper and the contribution of the author of this thesis to those papers.*

Contents

Part I.

Introduction and Background

1. Introduction

Aware of the threat it poses, detection of malware has been extensively researched in industry and academia alike. Despite being largely proprietary, antiviral software companies are believed to continue to rely on semi-automatic, signature-based processes to analyze and detect malicious applications (hereafter apps). That is, in addition to automated tools that attempt to identify the nature of an app (i.e., malicious or benign), human analysts are often summoned to assess the malignancy of apps under test, especially if the aforementioned tools fail to reach a verdict [98]. If an app is found to be malicious, a representation of this app (e.g., cryptographic hash digest, Control Flow Graph (CFG), or call graph), is generated and used as a reference against which representations of new apps discovered in the wild (e.g., on app marketplaces), are compared. Unfortunately, this process suffers from two limitations. Firstly, albeit necessary to ensure the accuracy of the labels assigned to apps (i.e., malicious or benign), involving humans in the analysis process significantly slows it down and renders it unable to cope with the continuous release of malicious apps. Secondly, relying on signatures, regardless of their type, confines the detection process only to either malicious apps that have been encountered before or ones that are very similar to them.

To address such limitations, researchers attempt to devise detection methods that (a) fully automate the malware analysis and detection processes presumably utilized by antiviral software firms, and (b) can detect malicious apps beyond its repository of previously-analyzed and detected apps. To that end, researchers have devised a plethora of detection methods that attempt to achieve these two goals [115, 139, 145, 131, 132, 79]. One particular method that gained popularity within the research community is ML-based malware detection [142]. ML algorithms excel at automatically identifying patterns and common characteristics shared by data points within a given dataset and recognize such patterns in data points beyond such a dataset (i.e., out-of-sample data points) [21]. These two properties of ML algorithms are what researchers rely on to implement ML-based detection methods that automatically learn characteristics of previously-analyzed malicious apps and use the learned characteristics to recognize the malignancy of out-of-sample ones.

Although researchers report promising preliminary detection results [44], only a few of commercial antiviral software are known to utilize ML-based detection [31, 64, 103]. Given the continuous threat of malware [144, 150, 96] and the dire need for unorthodox detection methods, the slow adoption of ML-based malware detection implies their inadequacy. In fact, some researchers have recently been calling for a reboot of the malware detection research area altogether [74]. The perplexing fact is that developers of different detection methods usually report promising results in their papers using real-world malware, which raises the question: how can a given detection method perform well under one particular

setting (i.e., the one adopted during its evaluation), yet fail to replicate such performance under another?

1.1. ML-Based Malware Detection

Malware authors tend to focus on platforms and operating systems that are popular among their victims, regardless of their objectives, viz. exhibition of technical capabilities [105, 25], monetary profit [142, 147], or sheer vandalism. Given its ubiquity and the willingness of its users to acquire new apps, the Android platform embodies a lucrative target for malware authors. Consequently, Android has been continuously targeted by malware authors with 98% of mobile malware instances targeting the Android operating system, users, and app marketplaces [142, 80]. So, given the vital role mobile devices play nowadays, in this thesis, we focus on Android malware as a case study of malware analysis and detection.

To identify the potential reasons behind the inability of current ML-based Android malware detection methods to replicate their performance in controlled evaluation environments, we start by discussing the typical process of automated Android malware analysis and detection. The problem of Android malware detection–and malware detection in general–can be reduced to that of *matching* out-of-sample apps to either individual apps or to multiple apps that depict a malware family, such as `DroidKungFu`, a malware type, such as `Adware`, or to labels that capture their malignancy (i.e., malicious versus benign). The group of apps to which an app is matched dictates the class of the app. To perform this matching, detection methods have to rely on repositories of Android apps that have been pre-labeled. Unknown (malicious) apps are matched against apps in those repositories.

The process of automated ML-based Android malware detection can be abstracted, as seen in Figure 1.1. Android apps are usually shipped in Android Package (APK) archives. Such apps can be acquired (step **(1)**) by crawling Android marketplaces, from online repositories, or using services, such as *AndroZoo* [10], that continuously crawl app marketplaces for new apps and download them. We hereafter refer to such collections of APK archives as app *datasets*. The more significant task within this step is that of labeling the apps in the acquired dataset. Depending on the problem a researcher is attempting to solve, the labels might range from simple binary labels (e.g., malicious and benign), to more specific labels that depict the malware families or types of apps.

In essence, ML algorithms are mathematical models that process vectors of numerical features that represent data points. In step **(2)**, numerical features are extracted–and further refined (step **(3)**)–from the Android APK archives to represent each app as a feature vector. The feature vectors along with the labels assigned to each app are used in step **(4)** to *train* a ML algorithm into a *model*. Training a model tackles the first limitation of conventional malware detection, namely it automatically finds common characteristics within the numerical representations of the apps, and uses the identified characteristics to segregate apps into the different classes dictated by the labels in ($\hat{y}$) (e.g., malicious versus benign).

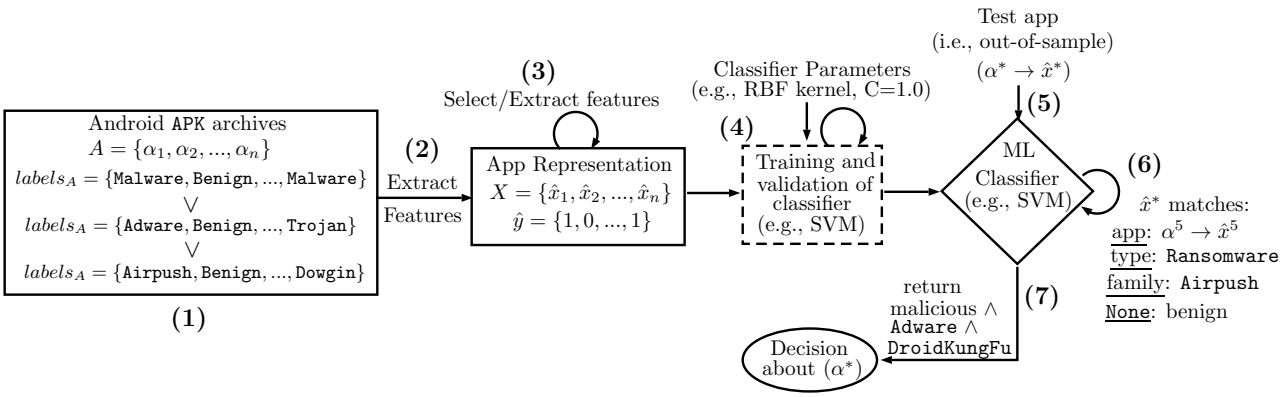
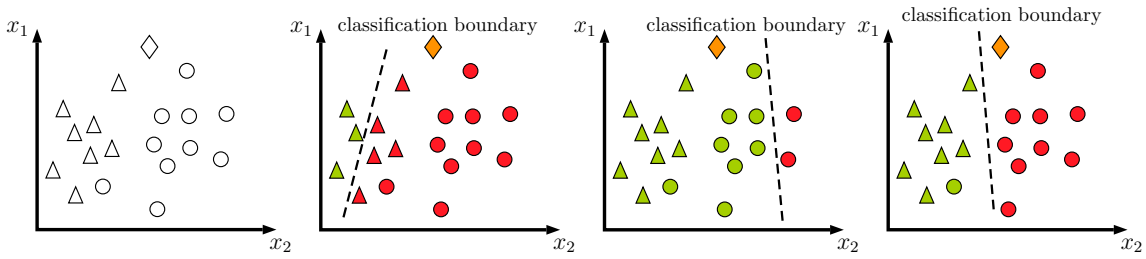


Figure 1.1.: A universal architecture of ML-based Android malware detection methods depicting the basic modules used to make decisions about apps' classes.

The main objective of ML-based detection method is to be able to accurately discern the classes of apps whose vectors were not used during the training phase (i.e., out-of-sample apps), effectively tackling the second and main limitation of conventional detection methods. Given a vector representation ($\hat{x}^*$) of an out-of-sample (or *test*) app (α^*) whose features mimics the ones extracted from its training counterparts, in step **(6)**, the trained ML model attempts to match ($\hat{x}^*$) to a class of feature vectors that share similar characteristics. In step **(7)**, the model returns the label depicting the class with which ($\hat{x}^*$) shared the most characteristics as the predicted label of (α^*).

The problems hindering the effectiveness of ML-based Android malware detection methods can dwell in any of the previous operations and modules. For example, extracting features from the APK archives of apps in the training dataset that do not reflect the class of the app (e.g., app package name), prevents ML algorithms from finding characteristics that can effectively segregate malicious and benign apps. Similarly, different ML algorithms are suited to different types and forms of features extracted from Android apps [72]. Hidden Markov Model (HMM), for instance, are used with traces of Application Programming Interface (API) calls made by apps during runtime [120], whereas numerical, embedded features are more suited for Support Vector Machine (SVM) [15]. In addition to these two problems of feature engineering and ML algorithm selection, the problem of accurate labeling is equally significant. Accurately labeling the apps in a dataset used by any detection method is fundamental to the quality and reliability of the decisions it makes. If apps are inaccurately or incorrectly labeled (e.g., malicious apps labeled as benign), regardless of the approach it adopts, the detection method will base its decisions on false information, effectively making false decisions about the nature of the test apps.

To further motivate the significance of correctly labeling apps used to train and evaluate ML-based detection methods, consider the illustration in Figure 1.2. Assume that we have a set of Android malicious (circles) and benign (triangles) apps that are represented using



(a) Unlabeled (b) Labeled with μ_1 (c) Labeled with μ_2 (d) Labeled with μ_3

Figure 1.2.: An illustration of the processes of labeling Android apps and using the labeled apps to train detection methods. Uncolored circles, triangles, and diamond depict malicious, benign, and unknown (zero-day) apps. Shapes colored in red indicate a malicious label, shapes colored in green indicate a benign label, and shapes colored in orange indicate an unknown label.

a two-dimensional feature set, as seen in Figure 1.2a. Now assume that we label these apps as malicious (red) and benign (green) according to three strategies: (μ_1), (μ_2), and (μ_3). The labels assigned to the apps are going to impact the decision boundary a detection methods (e.g., *SVM*), will attempt to find to separate apps into malicious and benign. The more accurate a labeling strategy (μ_i) is, the more precise a detection method can find a decision boundary between malicious and benign apps, which will impact the labels given to out-of-sample apps depicted here as an orange diamond. For example, the labeling strategy used in Figure 1.2b labels the apps in a manner that would put the orange diamond on the side of malicious apps. In contrast, using a labeling strategy that is more biased towards labeling apps as benign, the diamond will more likely be on the side of benign apps, as seen in Figure 1.2c.

In this hypothetical scenario, (μ_3) labels apps as malicious and benign more accurately than (μ_1) and (μ_2). However, this does not guarantee that the zero-day app is indeed malicious. Adopting a different feature set or a high-dimensional feature space may position the app's representation on a different side of the learned classification boundary. So, labeling strategies enable detection methods to be trained using more accurate labels and, in theory, be more effective. This does not impact the detection methods' abilities to detect apps far from what it has seen before (e.g., because they are newer [44]). As we discuss later in this chapter, adopting different labeling strategies might create different perspectives through which researchers assess the effectiveness of their ML-based detection method differently. For instance, albeit using the same training dataset and the same ML-based detection method, researchers adopting (μ_1) to label apps in Figure 1.2a will encounter different detection performance from their detection method than those researchers adopting (μ_2), which will discourage the latter group of researchers from further pursuing an otherwise promising approach to detect Android malware. Consequently, in order to objectively be able to evaluate the effectiveness of a given ML-based detection method,

researchers need to ensure that the labels they assign to apps used to train and evaluate the detection method reflect the ground truth of apps (i.e., malicious and benign), as accurately as possible.

Given the importance of labeling, in theory, labels should be assigned to apps after manual static and dynamic inspection of the apps' structures, codebases, and runtime behaviors. Nonetheless, the frequent release and discovery of malicious apps and the need of a large number of apps to train effective ML-based detection methods render manual labeling infeasible. As a workaround, the research community relies on the labels provided by online platforms with `VirusTotal` being the de facto platform to acquire and label apps [150, 75, 139, 153, 156]. `VirusTotal` does not label apps as malicious and benign. Given a hash of an app or its executable, the platform returns the labels given by around 60 antiviral scanners to the app along with information about its content and runtime behavior. It is up to the platform's user to decide upon strategies to interpret such information to label apps as malicious and benign.

1.2. Problems and Literature Gaps

In this section, we discuss the problems associated with relying on `VirusTotal` to label Android apps used in training and evaluating ML-based malware detection methods. We also discuss how the research community addresses each problem, effectively revealing gaps in the literature we aspire to fill.

We identified four main problems associated with relying on `VirusTotal` to label Android apps. First and foremost, relying on online platforms, such as `VirusTotal`, to generate the ground truth of Android apps imposes an **upper bound** on the performance of (ML-based) detection methods. To explain this, consider the following example: a researcher downloads the `VirusTotal` scan reports of Android apps in a dataset they use to test the detection performance of their newly-devised ML-based malware detection method. Regardless of how the researcher interprets the scan results in the aforementioned scan reports to label apps as malicious and benign, the generated labels act as the apps' ground truth. By definition, the labels predicted by the researcher's ML-based malware detection method cannot surpass the generated ground truth, even if such a ground truth is inaccurate. The predicted labels will, at best, mimic the `VirusTotal`-based ground truth indicating a perfect classification accuracy. As we discuss later, the scan result in `VirusTotal` scan reports can be interpreted in multiple ways and may in fact be flawed. However, regardless of the correctness of `VirusTotal` scan reports and how they are interpreted, this problem still persists: no detection method can perform better than its ground truth, which is based on `VirusTotal` in this case. Unfortunately, there is no clear solution to this problem, and we do not attempt to find a solution for it in this thesis. Instead, we accept this problem as a limitation of utilizing `VirusTotal` to label Apps, and we focus on providing the research community with insights about how to best utilize `VirusTotal` and interpret its scan reports to generate ground truth for Android apps that might better reflect their nature (i.e.,

malicious and benign).

Secondly, VirusTotal is confined to the labels given by commercial antivirus scanners. In some cases, such scanners are unable to detect the malignancy of an Android app (e.g., zero-day malware). Using VirusTotal to label apps will result into mislabeling this malicious app as benign, regardless of the utilized strategy to label the app. It follows that any ML-based detection method that is trained using such an incorrect label would mislabel similar malicious apps (e.g., belonging to the same malware family), as benign as well. Correctly labeling the mislabeled malicious app as malicious is, from the perspective of the ML-based detection methods, a false positive. So, we can state the first problem and literature gap with using VirusTotal to label Android apps as follows:

Using VirusTotal to label apps: Literature Gap 1

Upon evaluating novel ML-based detection methods using datasets labeled using VirusTotal, the performance of such methods is bound to the correctness of the labels given by VirusTotal's scanners, which cannot always provide accurate labels that reflect the ground truth of Android apps. In the literature, there are no clear suggestions on how to work around this problem via, for instance, identifying VirusTotal scan reports that may contain inaccurate verdicts and, hence, need to be avoided.

Recall that the role of platforms, such as VirusTotal, is not to help ML-based detection methods to correctly classify previously-analyzed apps; but rather to train models that address the limitations of conventional malware detection methods by detecting out-of-sample malicious apps. However, if the model is trained using inaccurate labels, it is unlikely that it could effectively separate malicious and benign apps and, in turn, detect out-of-sample malicious apps. Unfortunately, any platform that relies on the verdicts of commercial scanners, including VirusTotal, cannot circumvent this problem. The only other option is to either manually label apps, which is infeasible, or embed human analysts within the labeling process as suggest in [91]. In order to address this problem in this thesis, we break this problem into the following subproblems, addressing which could mitigate the negative impact of the main problem on the accuracy of labeling strateges. An important assumption in this thesis is that VirusTotal scan reports improve (i.e., in terms of the accuracy of labels given by different scanners), and stabilize over time. If we can find indications of stability of VirusTotal scan reports (e.g., age of scan report), then we can focus on apps whose scan reports are stable and are more likely to help accuractely label their corresponding apps. In the literature, there are subjective claims that VirusTotal scan reports stabilize within a specific period of time. Nevertheless, such claims neither define what is meant by stability nor the method used to estimate this period. That is, there are no clear methods in the literature to estimate the stability of a VirusTotal scan report.

Using VirusTotal to label apps: *Literature Gap 1.1*

In the literature, there are no methods to define or estimate the stability of a VirusTotal scan report, which can help avoid scan reports that do not reflect the malignancy of some Android apps.

The second subproblem focuses on the types of Android apps whose scan report usually take longer time to stabilize. The malignancy of some malware types, such as Adware, is often debated, which might be reflected in the number of scanners deeming them as malicious. In addition to some malware types, our experience with scanning in-house developed malicious apps with VirusTotal suggests that the scanners on the platform are oblivious to malicious apps that cannot be found in the wild. Lastly, commercial antivirus software usually need some time to recognize the malignancy of newly-developed malware. So, the VirusTotal scan reports of new malware instances are expected not to reflect the malignancy of such apps. In addition to these examples, identifying the types of apps whose VirusTotal scan reports can be misleading can help researchers to rule out those apps. Similar to the first subproblem, there are no methods in the literature that reveal such apps.

Using VirusTotal to label apps: *Literature Gap 1.2*

In the literature, there are no clear insights on how to identify Android apps whose VirusTotal scan reports might not reflect their ground truth.

In [44], Pendlebury et al. demonstrated the impact of time on the performance of ML-based detection method. In what is known as performance decay over time, they demonstrated that if a detection method exclusively relies on older apps, then it will eventually lose the ability to accurately classify apps that are more recently developed. So, ML-based detection methods should be continuously updated with new Android apps whose VirusTotal scan reports are yet to stabilize or those that belong to some malware types that are not recognized by all antivirus software as malicious. To be able to label such new apps, one can rely on the verdicts given by a subset of scanners that proved to be more accurate than others. However, this presumes that there is a universal and permanent set of VirusTotal scanners that are accurate across datasets. Given the dynamicity of VirusTotal, one cannot assert the existence of such a universal set of scanners. To the best of our knowledge, there are neither studies that identify such a universal set of scanners nor ones that define a methodology on how identify them.

Using `VirusTotal` to label apps: *Literature Gap 1.3*

In light of `VirusTotal`'s dynamicity, there are no studies in the literature that argue for or against the existence of a permanent set of `VirusTotal` scanners that are universally correct across different datasets.

The third main problem associated with using `VirusTotal` to label Android apps is that there are *no standard procedures* for interpreting the scan results acquired from `VirusTotal` to label apps. Researchers hence use their intuitions and adopt ad hoc threshold-based strategies to label the apps in the datasets used to train and evaluate their detection methods or release to the research community as benchmarks. In essence, threshold-based labeling strategies deem an app as malicious if the number of antiviral scanners labeling the apps as malicious meets a certain fixed threshold. For example, based on `VirusTotal`'s scan reports, Li et al. labeled the apps in their *Piggybacking* dataset as malicious if at least one scanner labeled them as malicious [75]. Pendlebury et al. labeled an app as malicious if four or more scanners did so, and based the evaluation of their tool on such threshold [44]. Wei et al. labeled apps in the *AMD* dataset as malicious if 50% or more of the total scanners labeled an app as such [153]. Finally, the authors of the *Drebin* dataset [15] labeled an app as malicious if at least two out of ten scanners they manually selected courtesy of their reputation (e.g., `AVG`, `BitDefender`, `Kaspersky`, and `McAfee`) did so. Using different thresholds significantly hinders comparing the performances of ML-based detection methods, even if they utilize the same dataset of training apps.

Using `VirusTotal` to label apps: Literature Gap 2

The lack of standards on how to interpret `VirusTotal` scan results to label (Android) apps forces researches to adopt subjective threshold-based labeling strategies to label apps which might hinder the comparison of different ML-based detection methods. Moreover, researchers neither specify the reasons behind adopting a particular threshold nor report the performance of their methods using different thresholds found in the literature.

Some of the aforementioned labeling strategies may indeed accurately label apps better than others. However, researchers have found `VirusTotal` to be a dynamic platform that frequently changes [101, 91, 92]. Unfortunately, the details of `VirusTotal`'s dynamicity and how it impacts the accuracy of threshold-based labeling strategies are not detailed in the literature. Without knowing how `VirusTotal` changes, researchers can neither devise labeling strategies that are more resilient to the platform's dynamicity nor design

and implement alternative platforms that avoid VirusTotal's limitations.

Using VirusTotal to label apps: Literature Gap 3

VirusTotal is known within the research community to be a dynamic, volatile platform. This dynamicity undermines the usage of fixed thresholds to label apps for prolonged periods. However, no significant details could be found in the literature about the aspects of its dynamicity, how it impacts the labeling accuracies of threshold-based labeling strategies, or how it can be circumvented.

The lack of solutions to the aforementioned problems associated with using VirusTotal to label Android apps used to train ML-based detection methods makes it difficult to objectively assess their performance. On the one hand, using a threshold-based labeling strategy that does not accurately reflect the ground truths of apps in a dataset used to train ML-based detection methods might negatively affect its performance, which leads their developers to dismiss a rather promising detection approach. On the other hand, developers of inadequate detection methods might get a false sense of confidence in the detection capabilities of their detection methods because they perform well, albeit using an inaccurate labeling strategy [100, 121]. Furthermore, the dynamicity of VirusTotal prevents the utilization of a fixed threshold for prolonged periods. So, if a ML-based detection method was evaluated using apps labeled based on VirusTotal scan reports and a certain threshold at one point in time, it is unlikely that the same threshold can be used in the future to replicate the initial evaluation results on a newer version scan reports.

1.2.1. Motivating Example

In this section, we give an example of how the problems associated with using VirusTotal to label Android apps manifest, and how they undermine the objective evaluation of a malware detection methods. During the evaluation of an Android malware detection method we implemented, we came across a dubious scenario involving a test app called TP.LoanCalculator that is part of the *Piggybacking* [75] dataset. Despite being labeled by the dataset authors as malicious, our detection method deemed this test app[1] as benign because it had the same metadata (e.g., package name and description), compiler, and even codebase digest as one benign app[2] that our detection method kept in a repository of reference benign apps.

Authors of the *Piggybacking* dataset labeled apps with the aid of VirusTotal scan reports using a threshold of one scanner [75]. After querying VirusTotal for the scan reports of both apps, we found that the test app was labeled malicious by 14 out of 60 antiviral software scanners, whereas all scanners that deemed the reference app as benign, which coincides with the authors' labels. However, we noticed that the scan reports acquired from

[1] 2b44135f245a2bd104c4b50dc9df889dbd8bc79b
[2] d8472cf8dcc98bc124bd5144bb2689785e245d83

VirusTotal indicated that the apps were last analyzed in 2013. So, we submitted the apps' APK archives for re-analysis in November 2018 to see whether the number of scanners would differ. After re-analysis, the malicious test app had three more scanners deem it malicious. More importantly, the number of scanners deeming the benign reference app as malicious changed from zero to 17 after re-analysis. Ultimately, we found that the reference app initially labeled and released as part of the *Piggybacking* dataset as a benign app is, in fact, another version of a malicious app of the type Adware. The authors of *Piggybacking* did not intentionally mislabel apps. The most likely scenario is that, at the time of releasing the dataset, the reference app was still deemed as benign by the VirusTotal scanners. This behavior demonstrates the problem with using VirusTotal that is associated with the first gap: the detection methods trained using VirusTotal-based labels cannot outperform the labeling accuracy of VirusTotal itself especially on previously-analyzed and labeled apps.

The numbers above mean that, at most, about 28.33% of VirusTotal scanners (i.e., 17 out of 60), deemed both apps as malicious with some renowned scanners, such as AVG, McAfee, Kaspersky, Microsoft, and TrendMicro continuing to deem both apps as benign. So, depending on the adopted labeling strategy, both apps may be correctly classified as malicious. For example, according to the dataset authors' strategy to label an app as malicious if at least one scanner deems it so [75], both apps would be labeled as malicious. The same would not hold for the authors of the *AMD* dataset who consider an app as malicious if at least 50% of the VirusTotal scanners deem it malicious [153]. This demonstrates the impact of lacking a standard methodology to devise thresholds that interpret VirusTotal scan reports to accurately label apps as malicious or benign (i.e., the problem associated with the second gap).

Lastly, to demonstrate the problem with using VirusTotal to label Android apps resulting from the third gap, we reanalyzed both apps six months later (i.e., in April 2019), to check whether the number of VirusTotal scanners deeming them malicious changed. We found that the number of scanners deeming the test and reference apps as malicious decreased from 17 to 11 and 10 scanners, respectively. Nevertheless, the number of scanners deeming both apps malicious increased in July 2020 to be 14 scanners. So, depending on the point in time the VirusTotal scan reports of both apps were acquired, a threshold-based labeling strategy using a threshold, such as 12 scanners, would assign different labels to the apps, which might impact the performance of the detection method relying on those labels during training.

The literature gaps we discussed in Section 1.2 cause problems with using VirusTotal to label Android apps used to train and evaluate ML-based detection methods. In this section, we demonstrated these problems and how they impact the performance of such methods. In the following sections, we detail the objectives of this thesis and how they attempt to devise solutions to such problems, effectively addressing the associated literature gaps.

1.3. Thesis Overview

The infeasibility of manually analyzing and labeling Android apps forces the research community to use VirusTotal scan reports to label those apps. In the previous sections, we discussed and demonstrated the problems associated with using VirusTotal to label apps and how it undermines the objectivity of evaluating ML-based detection methods meant to detect out-of-sample malicious apps. The main problems we discussed were related to VirusTotal's dynamicity, which is *not sufficiently discussed within the research community*, that makes it difficult to devise labeling strategies that cope with such a dynamicity. In this context, the ultimate solution to VirusTotal's dynamicity is to replace it with a more stable platform. However, until either VirusTotal addresses its limitations or an alternative platform is developed and tested, the research community is expected to continue to use VirusTotal. Furthermore, without revealing the aspects of VirusTotal's dynamicity, the research community risks implementing an alternative platform that suffers from similar shortcomings as VirusTotal. So, the **overarching objective** of this thesis is to provide the research community with actionable insights about VirusTotal, the aspects of its dynamicity, its limitations, and how to optimally interpret its scan reports to label Android apps accurately.

1.3.1. Research Questions

For each problem and corresponding literature gap we identified with using VirusTotal scan reports to label Android apps used to train and evaluate ML-based detection methods (detailed in Section 1.2), we posulate a number of research questions. Addressing these questions should help us solve or address their corresponding problem and, ultimately, achieve the main objective of this thesis, viz. optimally utilizing VirusTotal to label Android apps.

With the first three research questions, we attempt to address the first literature gap associated with using VirusTotal to label apps. Answering those research questions should enable us to identify Android apps whose VirusTotal scan reports might not reflect their ground truth. By ruling those scan reports out or devising special techniques to interpret them to label the corresponding apps, we can improve the quality of labels based on the VirusTotal scan reports.

RQ1: How can we deem a VirusTotal scan report of an Android app as stable before using it to label the app? (addressed in Chapter 5)

RQ2: What are the properties of an Android (malicious) app (e.g., malware type, age, source), that makes it difficult for VirusTotal scanners to correctly label it? (addressed in Chapter 2 and Chapter 4)

RQ3: Is there a universal ensemble of VirusTotal scanners that are more correct over time than others? (addressed in Chapter 5)

The fourth research question is linked with the second literature gap: the lack of standard methodologies to interpret VirusTotal scan reports in order to label Android apps. Regardless of the adopted labeling strategy, standardizing the interpretation of VirusTotal scan reports is expected to eliminate subjective, ad hoc threshold-based labeling strategies widely-adopted within the research community and, in turn, facilitate comparing the performance of different ML-based detection methods.

RQ4: How can we standardize the interpretation of VirusTotal scan reports to produce accurate labels for Android apps? (addressed in Chapter 4, Chapter 5, and Chapter 6)

We address the third literature gap of using VirusTotal by postulating the following research question. The literature mentions that VirusTotal is a dynamic platform, but does not detail its aspects of dynamicity and how they impact labeling strategies. Identifying those aspects enables us to devise labeling strategies or amend existing ones to circumvent them or minimize their negative impact. This might also contribute to the standardization of interpreting VirusTotal scan reports to accuractely label apps.

RQ5: What are the aspects of VirusTotal's dynamicity that impact the performance of labeling strategies, particularly threshold-based ones? (addressed in Chapter 4, Chapter 5, and Chapter 6)

Ultimately, VirusTotal needs to be amended or replaced altogether with a more accurate and stable platform. Without knowing the exact limitations of VirusTotal, the research community risks implementing alternative platforms that suffer from the same shortcomings. But answering the following research questions, we should be able to identify the main limitations of VirusTotal that need to be avoided. The answer to this research question is only made possible by answering the previous ones.

RQ6: What are the limitations of VirusTotal and how can they be mitigated? (addressed in Chapter 7)

1.3.2. Solutions

Our solution to answer the research questions postulated in the previous section is two-fold. Firstly, we studied the performance of different threshold-based labeling strategies based on VirusTotal scan reports over time. This study revealed an important aspect of VirusTotal's dynamicity (**RQ5**): the platform frequently and randomly changes the sets of scanners it includes in the scan reports of apps, including scanners that correctly label those apps. This finding implies that thresholds that were once found to yield accurate labels cannot be fixed and utilized for prolonged periods. Instead, before labeling Android apps based on their VirusTotal scan reports, the currently optimal thresholds need to be re-calculated. Given the popularity and simplicity of threshold-based labeling strategies, we devised an algorithm to identify the currently optimal thresholds at any point in time

and discussed the conditions that need to be satisfied in order for such an algorithm to be effective (**RQ4**).

Secondly, we implemented and evaluated a framework, Maat[3] that supports two main functionalities. Maat is a set of tools that automatically analyze a the VirusTotal scan reports of pre-labeled apps to (a) calculate the labeling accuracy of different VirusTotal scanners against different malware types and families over time (**RQ2**), (b) identify the set of VirusTotal scanners that proved to be the most accurate at labeling apps in any given dataset over time (**RQ3**), (c) find the set of VirusTotal scanners that whose labels do not fluctuate over time (i.e., stable scanners) (**RQ3**), and (d) trace and plot the evolution of the number of scanners deeming apps in a dataset as malicious as part of estimating the time take for a VirusTotal scan report to stabilize (**RQ1**). More importantly, instead of having to re-calculate the optimal threshold every time researchers wish to label apps in their datasets or rely on subjective, ad hoc thresholds, Maat offers an automated and systematic method to infer adequate labeling strategies as follows. Maat uses the information gathered from analyzing VirusTotal scan reports of apps in a pre-labeled dataset to automatically train ML-based labeling strategies that can withstand VirusTotal's dynamicity for longer periods of time and, hence, need not to be re-trained on a regular basis (**RQ4**). Our evaluation of Maat shows that its ML-based labeling strategies can match and outperform threshold-based labeling strategies that use the currently optimal thresholds at accurately labeling Android apps based on their VirusTotal scan reports. Furthermore, the ML-based detection methods whose feature vectors were labeled using Maat's ML-based labeling strategies outperformed those methods whose feature vectors were labeled by threshold-based labeling strategies. That is, Maat contributed to training ML-based detection methods more effective at detecting out-of-sample (malicious) apps.

1.3.3. Contributions

At the end of this research, the following contributions are made:

- **Revealing the types of Android apps that are difficult for VirusTotal scanners to correctly label.** We reveal in this thesis that VirusTotal scan reports do not always provide the correct verdicts vis-à-vis the label of an app (i.e., malicious or benign), which would have negative impacts on labeling strategies that consider such scan reports as ground truth. In the literature, we found no guidelines on how to identify Android apps whose VirusTotal scan reports may include misleading scan results (the first literature gap). In Chapter 2, Chapter 4, and Chapter 5, we reveal some characteristics of Android apps that correlate with misleading VirusTotal scan results, including the type of a malicious app (e.g., Adware), how recently

[3]Maat refers to the ancient Egyptian concepts of truth, balance, harmony, and justice. Our framework builds ML-based labeling strategies that harmonize the labels given by different VirusTotal scanners to provide accurate and reliable labels to apps.

was it developed, and whether it can be found on public platforms, such as app marketplaces.

- **Identifying the aspects of `VirusTotal`'s dynamicity, their impact on threshold-based labeling strategies, and how to counter them.** The dynamicity of `VirusTotal` and its scan reports undermines the effectiveness of threshold-based labeling strategies, which continue to be used within the research community. In the literature, we found no clear description of the aspects of `VirusTotal`'s dynamicity and how they impact this type of labeling strategies (i.e., the third literature gap we identified). To address this gap, in Chapter 4 and Section 5.2, we discuss the elements of `VirusTotal`'s dynamicity that undermine the labeling accuracy of conventional threshold-based labeling strategies that rely on fixed thresholds, which are commonly used within the research community. Furthermore, we propose a method to find the current optimal threshold of `VirusTotal` scanners to be used by threshold-based labeling strategies, and enumerate the conditions needed to be satisfied in order for this method to be effective.

- **A systematic method to automatically analyze `VirusTotal` scan reports for accurate labeling and better detection of Android (malicious) apps.** Threshold-based labeling strategies are subjective, susceptible to `VirusTotal`'s dynamicity, and require frequent analysis of `VirusTotal` scan reports to identify the current optimal thresholds. Unfortunately, the alternative methods that were developed by the research community remain largely manual, are insufficiently evaluated to verify their impact on the malware detection process, and do not provide insights to the community on how or when to be used (i.e., the second literature gap associated with utilizing `VirusTotal`). We address this literature gap by implementing and publicly releasing (https://github.com/tum-i22/Maat) the source code of Maat (Chapter 5): a framework that provides the research community with a systematic method to generate ML-based labeling strategies on-demand based on the current scan results provided by `VirusTotal`. The results of our experiments show that Maat's ML-based labeling strategies are less sensitive to the dynamicity of `VirusTotal`, which enabled them to (a) accurately label apps based on their `VirusTotal` scan reports more consistently than their threshold-based counterparts and (b) improve the detection capabilities of ML-based detection methods (Chapter 6).

- **Detailing the limitations of `VirusTotal` and how to mitigate them.** There are voices within the research community that call for the replacement of `VirusTotal`. However, without a clear enumeration of the shortcomings of `VirusTotal`, we risk implementing alternative labeling platforms that suffer from the same shortcomings of `VirusTotal`. Revealing the aspects of `VirusTotal`'s dynamicity enabled us to identify **four** main limitations of `VirusTotal` that sometimes undermine its reliability and usefulness. Those limitations are (a) frequent, seemingly haphazard inclusion and exclusion of scanners in the scan reports of apps, (b) using inadequate

versions of scanners that are designed to detect malicious apps for other platforms, (c) refraining from frequently and automatically reanalyzing and re-scanning apps, and (d) denying access to the history of scan reports. Unless `VirusTotal` addresses such limitations, an alternative platform ought to be implemented and adopted by the research community. In Chapter 7, we propose a blueprint of an alternative scanning platform, Eleda, that is meant to provide the research community with more reliable and trustworthy labels.

In implementing the previously-discussed solutions, the following peer-reviewed publications either have been published or are under review:

1. **Salem, A.**; Paulus, F.; Pretschner, A. *Repackman: A Tool for Automatic Repackaging of Android Apps*. In Proceedings of the 1st International Workshop on Advances in Mobile App Analysis (A-Mobile), 2018.

2. **Salem, A.**; Pretschner, A. *Poking the Bear: Lessons Learned from Probing Three Android Malware Datasets*. In Proceedings of the 1st International Workshop on Advances in Mobile App Analysis (A-Mobile), 2018.

3. **Salem, A.**; Schmidt, T.; Pretschner, A. *Idea: Automatic Localization of Malicious Behaviors in Android Malware with Hidden Markov Models*. In Proceedings of the International Symposium on Engineering Secure Software and Systems (ESSoS), 2018.

4. **Salem, A.**; Hesse, M., Neumeier, J., Pretschner, A. *Towards Empirically Assessing Behavior Stimulation Approaches for Android Malware* . In Proceedings of the 13th International Conference on Emerging Security Information, Systems and Technologies (SECURWARE), 2019.

5. **Salem, A.**; Banescu, S., Pretschner A. *Maat: Automatically Analyzing VirusTotal for Accurate Labeling and Effective Malware Detection*. In ACM Transactions on Privacy and Security (TOPS) **[Under Review]**, 2021.

6. **Salem, A.** *Towards Accurate Labeling of Android Apps for Reliable Malware Detection*. In Proceedings of the 11th ACM Conference on Data and Application Security and Privacy (CODASPY), 2021.

In addition to the previously enumerated papers, the author of this thesis has co-authored the following peer-reviewed publications, which tackle relevant problems, related to the topic of this thesis, but are not part of this thesis:

7. **Salem, A** *GoldRusher: A Miner for Rapid Identification of Hidden Code*. In Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2018.

8. **Salem, A**; Banescu, S. *Metadata Recovery From Obfuscated Programs Using Machine Learning*. In Proceedings of the 6th Software Security, Protection and Reverse Engineering Workshop (SSPREW), 2016. **Best Paper Award**.

9. Banescu, S; Wuechner, T; **Salem, A**; Guggenmos, M; Ochoa, M; Pretschner, A. *A Framework for Empirical Evaluation of Malware Detection Resilience Against Behaviour Obfuscation*. In Proceedings of 10th International Conference on Malicious and Unwanted Software (MALWARE), 2015.

1.3.4. Datasets

In this section, we briefly discuss the composition and the role of the datasets we used in conducting our experiments to answer the research questions postulated in Section 1.3.1.

The largest dataset we use in this paper is a combination of 24,553 malicious apps from the *AMD* dataset [153] and 24,162 benign apps we downloaded from *AndroZoo* [10]. The malicious apps of *AMD* are meant to provide an overview of malicious behaviors that can be found in Android malware, spanning different malware families (e.g., `DroidKungFu`[161], `Airpush`[42], `Dowgin`[45], etc.) and different malware types (e.g., `Adware`, `Ransom(ware)`, and `Trojan`). To build the dataset, the authors of *AMD* only considered apps whose `VirusTotal` scan reports indicate that at least 50% of the scanners deem them as malicious, clustered them into 135 malware families, and manually analyzed samples of each family to ensure their malignancy. After analysis, the behavior of each family is represented as human-readable, graphical representation[4] of the behavior adopted by apps in each of the 135 malware families that can be found in the dataset. The involvement of human operators in labeling the apps and the high number of scanners deeming them malicious significantly decreases the likelihood of a benign app being mistakenly labeled as malicious. So, we consider all apps in the *AMD* dataset as malicious.

As for the benign apps we acquired from *AndroZoo*, we downloaded a total of 30,023 apps that were gathered from the Android official app store, *Google Play*. Google Play employs various checks to ensure the sanity of an app upon being uploaded [98], but sometimes malicious apps make it to the marketplace [150, 82]. So, we only considered apps whose `VirusTotal` scan reports indicate that **no** scanners deemed them as malicious at any point in time. This criterion does not guarantee that the apps' scan reports will continue to have a *positives* attribute of zero in the future. However, given that the apps were collected from Google Play, which already employs various checks to ensure the sanity of an app upon being uploaded to the marketplace [98], we presume that the `VirusTotal` scan reports of such apps will not radically change in the future in a manner similar to the apps discussed in Section 1.2.1. Consequently, we consider all apps in the *GPlay* dataset that fit the aforementioned criterion (i.e., 24,162 apps) as benign.

We refer to the combination of apps in the two previous datasets as *AMD+GPlay*, whose `VirusTotal` scan reports are used to train Maat's ML-based labeling strategies. We

[4]Example: `Airpush` family's first variety (http://tiny.cc/34d86y)

Table 1.1.: A summary of the datasets we utilize in this thesis, their composition, their sources, and the experiments within which they are used.

Dataset Name	Total Apps	Source	Usage
AMD+GPlay	24,553 (malicious)	*AMD*'s Website	◦ Training ML-based labeling strategies (Chapter 5) ◦ Demonstrating degrees of malignancy (Section 2.5)
	24,162 (benign)	*AndroZoo*'s servers	◦ Calculating scanner correctness (Section 5.2) ◦ Estimating scan report stability (Section 5.3)
AndroZoo	6,172 (unlabeled)	*AndroZoo*'s servers	◦ Training ML-based detection methods (Chapter 6)
Hand-Labeled	100 (76% benign+24% malicious)	*AndroZoo*'s servers	◦ Testing accuracy of labeling strategies (Chapter 4 and Section 6.1) ◦ Testing accuracy of ML-based detection methods (Section 6.4)
Hand-Labeled 2019	100 (90% benign+10% malicious)	*AndroZoo*'s servers	◦ Testing accuracy of labeling strategies (Chapter 4 and Section 6.1) ◦ Testing accuracy of ML-based detection methods (Chapter 4 and Section 6.4)

downloaded the existing scan reports of those apps and found that the overwhelming majority of the apps in the *AMD* dataset were last scanned in 2018, albeit spread across different months. So, we reanalyzed all of those apps in November 2018. Between April 12^{th}, 2019, and November 8^{th}, 2019, we reanalyzed all of the 53K apps and downloaded the latest versions of their `VirusTotal` reports every two weeks in accordance with Kantchelian et al.'s recommendations [63]. Aware of the fact that some apps in the dataset are as old as eight years, we attempted to acquire their older `VirusTotal` scan reports. Unfortunately, access to such reports is not available under academic licenses.

The second dataset we use is a random collection of 6,172 apps developed in between 2018 and 2019 and downloaded from *AndroZoo*. So, we refer to it as *AndroZoo* throughout this thesis. Unlike apps in *AMD+GPlay*, this dataset does not focus on a particular marketplace or class (e.g., malicious). This dataset is used in Chapter 6 to assess the ability of different labeling strategies to train more accurate detection methods. This dataset is meant to simulate the process of a researcher acquiring new Android apps and labeling them to train a ML-based detection method that detects novel Android malware, that `VirusTotal` scanners are yet to assign labels to. So, we use apps in this dataset to statically extract numerical features from their APK archives, represent them as feature vectors, and use them for training different types of machine learning classifiers. We use different threshold-based and ML-based labeling strategies to label those feature vectors prior to training the aforementioned classifiers.

The trained classifiers are then used to label apps in two small test datasets and compare the predicted labels with the ground truth. We refer to those small datasets as *Hand-Labeled*[5] and *Hand-Labeled 2019*[6]. Both datasets comprise 100 Android apps that were downloaded from *AndroZoo* and *manually analyzed and labeled*, to acquire reliable ground truth. The exact process we adopted in analyzing and labeling those apps can be found online and in

[5]http://tiny.cc/95bhaz
[6]http://tiny.cc/a7bhaz

appendix A. The primary difference between both datasets is that apps in the latter were developed in 2019. We ensured that apps in both datasets do not overlap with apps in the *AMD+GPlay* and *AndroZoo* datasets. We manually analyzed such 200 apps to acquire reliable ground truth that depicts the apps' true nature.

1.4. Structure

The remainder of this thesis is organized as follows. Chapter 2 provides an overview of Android malware, its structure, the payloads and functionalities that it usually includes, the evasion techniques it implements, and the subjectivity of defining and labeling apps as malicios. Chapter 3 discusses the theory and practice of malware detection before focusing on ML-based detection methods, their building blocks, and the challenges that face them. In Chapter 4, we delve into studying threshold-based labeling strategies and how `VirusTotal`'s dynamicity impacts their performance. Chapter 5 introduces Maat, the method we implemented to automatically analyze `VirusTotal` scan reports and devise a ML-based labeling strategy. Chapter 6 is dedicated to evaluating the applicability of Maat according to three criteria: (a) the ability of Maat's ML-based labeling strategies to accurately labeling Android apps according to their `VirusTotal` scan reports, (b) whether the trained ML-based labeling strategies can contribute to enhancing the performance of ML-based malware detection methods and enable them to detect zero-day malicious apps that `VirusTotal` scanners fail to recognize as malicious, and (c) whether and how long can the trained ML-based labeling strategies can withstand `VirusTotal`'s dynamicity to accurately label Android apps. Chapter 7 summarizes the limitations of `VirusTotal` we identified throughout this thesis and discussed methods to mitigate them. Chapter 8 presents related work. Chapter 9 presents conclusions, insights, and future work.

2. Android Malware

This chapter presents fundamental concepts about Android apps and the process of implementing malicious and potentially unwanted versions of such apps. The chapter then presents an overview of the trends and functionalities adopted by Android malware found in practice including the nature of the payloads found in them. Lastly, the chapter presents the subjectivity of labeling malicious apps belonging to different malware types. Parts of this chapter have previously appeared in peer-reviewed publications [8], [118], and [119] co-authored by the author of this thesis.

The Oxford dictionary defines malware[1] as "Software that is specifically designed to disrupt, damage, or gain unauthorized access to a computer system" [73]. Practically and more technically, the security firm `MalwareBytes` [85] defines malware as "malicious program or code that can **steal**, **encrypt**, or **delete** your data, **alter** or **hijack** core computer functions, and **spy** on your computer activity *without your knowledge or permission*." The majority of other definitions might utilize different terminologies but still revolves around the same notion of jeopardizing the confidentiality, integrity, and availability of a device, the apps installed on it, and the data it stores.

To increase the likelihood of infecting more systems, authors of malware usually target platforms and systems that are more popular among users. Given its popularity and ubiquity, the *Android* operating system has been targeted more often by malware [150, 153]. Consequently, we focus on Android malware as a case study in this thesis. In this thesis, we build on the previous definitions to define Android malware as follows:

Definition

Android malware are Android apps that are deliberately implemented to jeopardize the confidentiality, integrity, and/or availability of the device (including hardware components), the apps and services it contains, and the (personal) data it holds with or without the knowledge of the user.

Similar to other types of malware, Android malware continuously evolves to evade detection by malware detection methods [118, 142, 53, 158]. However, given the richness of the Android API and the sensitivity of some of the data stored on user devices (e.g., contacts,

[1] Malware is a portmanteau of the words malicious and software referring to software that is designed to damage a computer or a network of computers [90].

personal photographs, passwords, and so forth), the functionalities of Android malware differ, which might lead to different views of what constitutes a malicious functionality. In this chapter, we discuss the structure of Android malware in Section 2.2, the common functionalities and payloads it usually contains in Section 2.3, and the common techniques it employs to evade detection and hinder analysis in Section 2.4. Lastly, in Section 2.5, we discuss how the different structures and functionalities of Android malware, along with the subjectivity of deeming Android apps as malicious create more consensus among antiviral software vis-à-vis the malignancy of some malware families and types than others. However, prior to delving into defining Android malware and discussing its structure, we briefly discuss the structure of Android apps in general, its components, compilation process, and how it is distributed in Section 2.1.

2.1. Android Apps

2.1.1. Components

Android apps are primarily written in `Java` and are typically divided into four main components, viz. activities, services, broadcast receivers, and content providers [39]. Each of the aforementioned components–written as `Java` classes–serves a specific purpose that contributes to the overall functionality of the app. Activities depict the user interface element of an app. An activity is a screen containing Graphical User Interface (GUI) elements (e.g., `Button`), that users interact with to carry out a particular task (e.g., sending an email). Each Android app must declare an activity, referred to as the *Main* activity, that is displayed to the user once the app is started. In other words, the Main activity is the app's main entry point.

Services are components that carry out background tasks on behalf of the user (e.g., updating an app, downloading a resource, and playing music). The rationale behind this type of component is to execute long-running tasks without blocking or interrupting the user's interaction with other activities. In fact, without completing its task or being explicitly stopped, services can run in the background even if an app is terminated. Consequently, services do not have any GUI elements and are, hence, oblivious to the user. Services are usually explicitly started using the `startService` method.

Broadcast receivers, or simply receivers, are components that are meant to respond to broadcast messages or notifications from other apps or from the Android system itself. For example, after they complete a task, services can broadcast a message to inform an app of the task's completion. The Android system captures such broadcasts and forwards them to app components that *register* or listen to them, namely broadcast receivers. Receivers can listen to system broadcasts by declaring them as `intent-filters`. Such broadcasts can be system broadcasts, such as `android.intent.action.PHONE_STATE` which intercepts incoming phone calls, or custom ones, such as `com.myapp.CUSTOM_INTENT`.

Listing 2.1.: Building an implicit `Intent` object to send an email.

```
Intent email = new Intent(Intent.ACTION_SEND, Uri.parse("mailto:"));
email.putExtra(Intent.EXTRA_EMAIL, recipients);
email.putExtra(Intent.EXTRA_SUBJECT, subject.getText().toString());
email.putExtra(Intent.EXTRA_TEXT, body.getText().toString());
startActivity(Intent.createChooser(email, "Choose an email client from..."));
```

The primary method used by the system and Android apps to communicate is via `Intent`s. Intents are objects whose attributes carry information to other apps. For example, as shown in Listing 2.1, to send an email using a mailing client app, an intent object is populated with the list of recipients, the email's subject, and the email's body before being forwarded to the mailing client app. There are two types of intents, viz. explicit and implicit. On the one hand, explicit intents target specific apps. Apps use this type of intents to start other components, usually within the same app. In Listing 2.2, the `TargetActivity` activity is started using an explicit intent. On the other hand, implicit intents do not specify target components. Instead, they rely on a parameter, referred to as `action`, that resembles an intent-filter declared by some apps, effectively narrowing down the apps that can respond to the broadcasted intent. Upon receiving this type of intents, the operating system retrieves all apps that register to that particular action and prompts the user to choose one of them to receive and process the intent. For example, in Listing 2.1, if a device has multiple mailing client apps installed (e.g., Gmail and K-9), the user will be asked to choose one app to receive the intent.

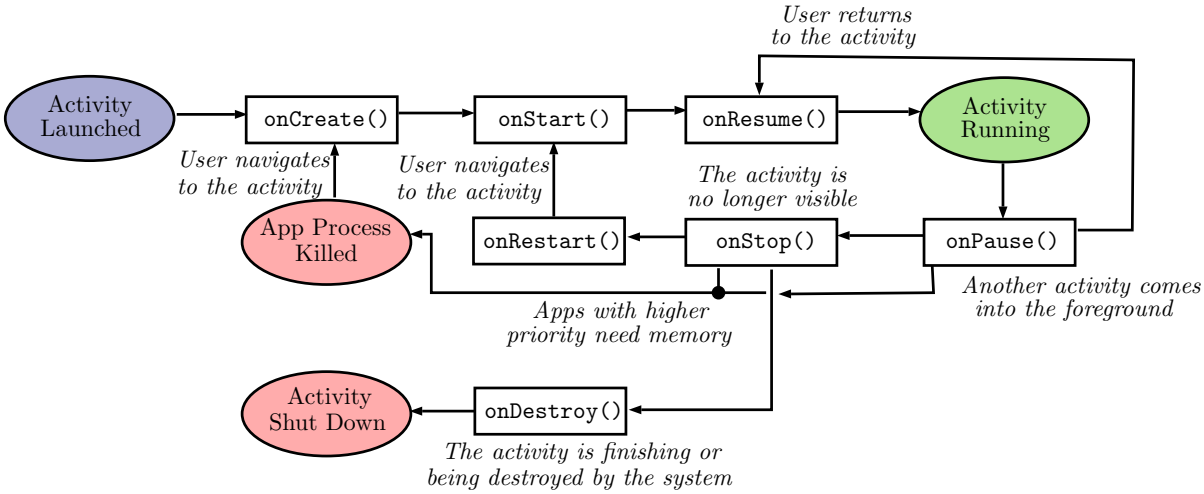
The ability to start different app components using intent objects means that Android apps do not have a single entry point. Unlike conventional `Java` programs, execution does not always have to start from the `main` method; it can start from a broadcast receiver which registers to incoming phone calls or Short Message Service (SMS) messages, for instance. This nature of Android apps makes them sometimes harder to analyze and, for instance, generate CFG's or call graphs that depict their structures [17].

Listing 2.2.: An explicit `Intent` to start `TargetActivity`

```
// Explicit Intent by specifying its class name
Intent i = new Intent(SourceActivity.this, TargetActivity.class);
startActivity(i);
```

The last main component of Android apps is content providers. Content providers abstract access to data structures that store app data, such as relational databases and configuration files. However, content providers are not the only way for Android apps to interact with local and remote data structures. From within other components, different libraries can be used to load and store data from and to text files, Structured Query Language (SQL) databases, and network sockets. For example, the package `android.database.sql`

Figure 2.1.: The typical life cycle of an activity in an Android app [33].



`ite` [35] enables Android apps to perform Create, Read, Update, and Delete (CRUD) operations with `SQLite` databases.

Being implemented in `Java`, the Android API follows an object-oriented architecture. Consequently, all of the aforementioned components are implemented as `Java` classes. When a new app is developed, its components have to inherit the pre-existing classes for activities, services, receivers, providers, and other subordinate components (e.g., Fragments). For example, all new activities inherit the `Activity` class, whereas all services inherit the `Service` class. The inherited classes contain different callback methods that respond to different stimuli (e.g., GUI events), that are usually overridden to accommodate for the apps' needs and use cases. Taking the `Activity` class as an example, the class contains methods such as `onCreate`, `onStart`, `onResume`, and `onStop`, and `onDestroy`. Respectively, these methods are executed whenever the activity is first created and its GUI elements being built, becomes visible to the user, is being interacted with by the user after a pause, and is sent to the background and no longer visible to the user. Developers of video playback apps, such as YouTube, probably need to override the `onStop` method to stop playing the video whenever the app is sent to the background, and override the `onResume` method to continue playing the video once the app is brought back to the foreground. Together these callback methods form what is known as a component's *life cycle* [39]. For example, Figure 2.1 shows the typical life cycle of an activity component.

Other components that Android apps usually contain include the android manifest file, resource files, and external libraries. Each Android app must have an Extensible Markup Language (XML) manifest file, called `AndroidManifest.xml`, that contains the following information. Firstly, the manifest file contains the app's package name (e.g., `com.company.app`), its version, and the minimum and maximum Android Software

Development Kit (SDK) versions it supports. Secondly, and more importantly, the manifest file enumerates the different components used by the app, their class names, and, if applicable, the intents they listen to. The manifest file declares the hardware and software features that the app requires to function properly. For example, an app like Snapchat needs to be installed on a device that contains a camera. Having mentioned system resources and features, the `AndroidManifest.xml` file declares the permissions the app needs to be granted to access different system resources. In the case of Snapchat, access to the device's cameras can only be granted if the app's manifest file contains the `android.permission.CAMERA`. Needless to say, different device resources have different degrees of sensitivity. Arguably, access to the device's camera is more sensitive than access to the internet, given the risk of being spied on by a malicious app. Consequently, permissions needed to access such sensitive resources are referred to *dangerous* permissions. Starting from Android 6.0 (API level 23), dangerous permissions need to be explicitly granted by users to apps the first time apps request access to a sensitive resource. However, regardless of their type, all permissions that can be potentially used by an app need to be declared in the `AndroidManifest.xml` file.

In essence, resource files comprise XML files that store information about the app's layout and miscellaneous values it uses during runtime. Layout resource files enumerate the GUI elements of app activities, including information about the type of each element, its position on the screen, texts it displays (e.g., a `Button` with the label *Login*), and callback methods it reverts to upon interaction. Layout XML files are retrieved during the creation of an activity, and the GUI elements they include are rendered and displayed to the user by the system. Often, the same colors and texts are repeatedly used by the aforementioned GUI elements. To avoid having to modify multiple layout files to change colors or fix a typographical error, these values are stored in resource files. For example, the strings and colors used throughout the app are usually stored in the `strings.xml` and `colors.xml` files and are easily accessible from, both, other XML files and `Java` classes.

Lastly, some apps might need direct, fast access to system resources. This can be achieved by writing libraries in `C/C++` using the Android Native Development Kit (NDK) and accessing them from the app's `Java` classes. For example, gaming apps usually defer rendering graphics to `C/C++` libraries for better performance. Those libraries are compiled and shipped with the apps' `Java`-based components as shared object libraries.

2.1.2. Compilation

The first step in compiling an Android app is to compile the `Java` components into `Java` Bytecode. This step is carried out in a manner similar to conventional `Java` applications, namely using the Java Development Kit (JDK). Optionally, tools such as *Proguard* are used to optimize and obfuscate the generated Bytecode [36]. Source code written in `C/C++` is compiled by the NDK's *CMake* to fit the architecture on which the app is to be deployed.

In most cases, Android developers implement their apps within an Integrated Development Environment (IDE)(e.g., eclipse or Android Studio), and use the default Android SDK to

compile their apps. If so, the generated, and possibly optimized, `Java` Bytecode is further translated using the SDK's `dx` compiler into `DEX`, short for Dalvik Executable, Bytecode and written into a single file called `classes.dex`. In some cases, developers slightly modify the app's codebases (e.g., to fix a bug), and recompile it. Only the affected components will be recompiled and merged with the original code using the `dexmerge` compiler, which also is part of the Android SDK. Other non-standard compilers are utilized to recompile an Android app from an intermediate representation, called `Smali`, after being disassembled using reverse engineering tools. For example, the reverse engineering tool Apktool [14] uses the `dexlib` compiler to recompile apps.

Albeit being optimized for Android, the `DEX` Bytecode cannot directly execute on Android devices. Prior to Android 4.4 (KitKat), Android systems adopted a model similar to that for `Java` applications, namely to use a Virtual Machine (VM) to fetch, compile, and execute Bytecode instructions in a *Just-In-Time* manner. On Android systems, this VM is called Dalvik Virtual Machine, which fetches `DEX` Bytecode from the `classes.dex` file translates them into machine code, and executes them, which slows down the execution of Android apps. Starting from Android 4.4, Android systems switched to the Android Runtime (ART) model, which compiles the entire Android app upon installation to machine code. This process is referred to as *Ahead-of-Time* compilation. However, the introduction of the new runtime system does not change the way Android apps are developed or compiled. That is to say, Android apps continue to be developed in `Java`, `C/C++`, and possibly `Kotlin` [95], and are compiled into `DEX` Bytecode.

2.1.3. Distribution

Android apps are distributed as ZIP archives that are referred to as APK. This APK archive contains the app's codebase in the `classes.dex` file and the shared object libraries, a compiled version of the `AndroidManifest.xml` file, the XML resources files, any external libraries used by the app (e.g., Java ARchive (JAR) or Dynamic-link Library (DLL) files), and the developer's certificate. As part of Google's policy, every Android app needs to be signed by its developer(s) [150]. However, this policy allows developers to generate their own certificates and self-sign their apps. In fact, researchers have found that 99% of Android apps are self-signed [11]

Signing an app is the last step before distributing it to users, usually via app marketplaces. App marketplaces are online platforms that host Android apps, allow users to download and install them, and provide a communication channel between users and app developers (e.g., via comments and feedback sections). The most renowned app marketplace is Google's official marketplace: Google Play. As of late 2018, Google Play hosted 2,031,946 apps [150], making it the largest marketplace for Android apps. Nonetheless, some countries like China block access to all Google services, including access to Google Play. So, owners of Android devices in those countries turn to third-party marketplaces to acquire new apps. Some of these marketplaces are Tencent, Baidu, AnZhi, and App China.

Depending on their intended target market, developers can upload their apps to one or

multiple marketplaces. By comparing the apps on Google Play and Chinese marketplaces, Wang et al. found that 77% of the apps published in Google Play are single-store ones (i.e., cannot be found on other marketplaces). A smaller percentage of apps were found to be published on multiple marketplaces, according to the same study. For example, between 20% and 30% of apps published in Chinese marketplaces can also be found on Google Play [150].

Upon being uploaded to marketplaces, apps are usually vetted to ensure they do not carry out any malicious functionalities. According to Wang et al., the majority of app marketplaces employ some type of app vetting; only eight marketplaces claim to incorporate human inspections to complement automated vetting processes [150]. The exact description of combining manual and automatic inspection of apps is usually proprietary and not made public. To circumvent this security by obscurity policy, researchers have attempted to fingerprint the vetting processes of marketplaces such as Google Play [98], and found that apps are automatically analyzed via static and dynamic analysis tools; if such tools fail to deem apps as benign, a human analyst is consulted to analyze an app.

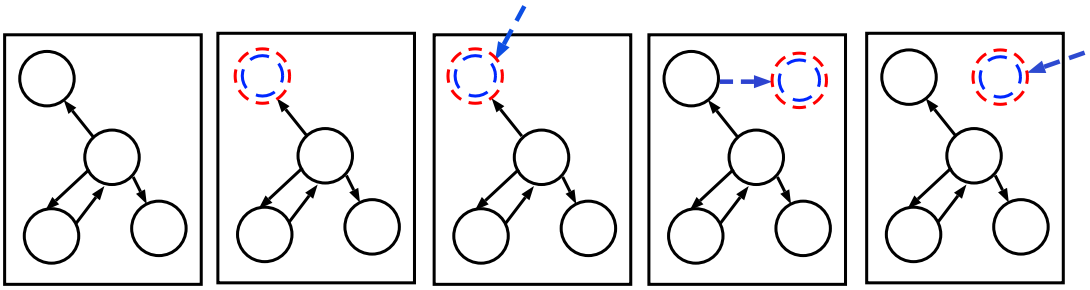
Despite such vetting processes, malicious apps still manage to evade detection and make it to the marketplaces [150, 119, 142, 75]. For example, Wang et al. found that at most, 17.03% of apps in Google Play were labeled as malicious by at least one of the antiviral scanners found on the online platform `VirusTotal`, whereas this number can rise to roughly 50% in case of Chinese marketplaces [150]. Consequently, even after the successful publication of apps, marketplaces continue to vet apps and remove them if they prove to be malicious. It was found that Google Play manages to remove about 84% of uploaded apps that are found to be malicious after publication, whereas Chinese marketplaces manage on average to remove 23.04% of those apps.

2.2. Structure of Android Malware

Android users are continuously in pursuit of apps to customize their devices or to meet a specific need (e.g., barcode scanning, audio recording, flashlight apps, etc.). Without implementing similar functionalities, Android malicious apps are unlikely to be downloaded and installed by users. That is, Android malware authors need to invest time to surround the malicious functionalities, or *payloads*, in their apps with benign ones that encourage users to download them.

There are two main methods to wrap payloads with benign code. Malware authors can implement both the malicious payloads and the benign code from scratch in what is known as *standalone* malware [153, 142, 161]. This model makes the malicious app more stable and gives its author more control over its behavior. However, it is a distraction for malware authors from their primary objectives of deploying malicious payloads on the users' devices. Furthermore, users tend to trust apps that are developed by renowned firms or developers and have been downloaded more often. To leverage the benefits of pre-existing trust in renowned apps and to avoid having to develop benign code from scratch, some malware

Figure 2.2.: Different methods a malicious payload can be added to an Android app and, if needed, triggered from within benign code segments. The dashed red nodes depict the components where the malicious payloads can be added, the blue dashed circles refer to possible triggers that control the execution of such payloads, and the blue arrows depict external triggers that might invoke the malicious payloads (e.g., system notification).



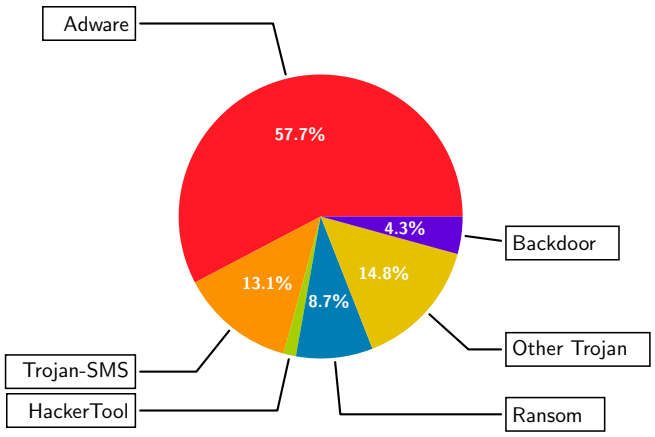
authors adopt repackaging as their model to implement malicious apps [118, 153, 160]. Repackaged malware [161] or piggybacked apps [75, 76] are benign apps (e.g., Angry Birds), that have been downloaded, decompiled, and grafted with malicious payloads. We discuss in Section 2.4.6 the exact technical details of one approach to repackaging Android apps.

Regardless of how the malicious payloads are added to an app, the structure of Android malware in terms of the relationship between the malicious and benign code segments can assume a finite number of forms, as seen in Figure 2.2. We refer to those forms as *deployment methods*. Assuming that subfigure (a) depicts the call graph of an Android app with the nodes depicting different components (e.g., activities), and the edges resembling how they invoke one another (e.g., via intents or direct method invocations). Subfigure (b) depicts the case in which the malicious payload is added to one of the pre-existing components, for example, as a new method or even as statements mingled with the original ones.

To partly conceal the injected payloads or control their execution, malware authors often wrap those payloads with conditional statements (subfigure (b)) or inject the payloads in components that register to external stimuli (subfigure (c)), such as receivers [75]. Those conditional statements and stimuli are referred to as *hooks* [75], *triggers* [118], or *schedulers* [8, 153]. We discuss different types of triggers and schedulers in Section 2.4.5.

Subfigures (d) and (e) resemble the scenarios in which the malware author opts to create a new component to host their malicious payloads. The newly-added component can be invoked either from the pre-existing components or from external sources, viz. other apps or the Android system itself. Although it avoids interfering with the app's existing code, deploying malicious payloads in separate components is less subtle, especially since they need to be declared at the `AndroidManifest.xml` file, as discussed in Section 2.1.1.

Figure 2.3.: The distribution of most common malware types of malicious apps found in Wei et al.'s *AMD* dataset [153].



2.3. Android Malware Payloads and Functionalities

Payloads implement the intentions of the malware authors. In theory, those intentions can include inflicting a monetary loss on users by sending SMS messages to premium numbers, deleting the user's contacts, leaking the current user's Global Positioning System (GPS) location, fingerprinting the device via International Mobile Equipment Identity (IMEI), etc. However, putting aside sheer vandalism and recognition within the malware authorship community, the majority of malware authors aspire to make monetary profit [140, 161].

The main source of profit for developers and marketplaces is via advertisements [2, 23]. Similarly, Android malware authors tend to develop payloads that generate profit from advertisements. Android malware that contains this type of payload is usually referred to as `Adware`. The main difference between `Adware` and benign Android apps that use advertising libraries and servers is that the former aggressively display advertisements to users and usually gathers sensitive information from the users' devices without their knowledge to display more targeted advertisements. Researchers have found that `Adware` is the most common malware type to be found in app marketplaces [119, 153]. For example, Wei et al. gathered a total of 24,553 malicious apps from different app marketplaces, of which 57.7% belonged to the `Adware` malware type. As seen in Figure 2.3, the second most common malware type in Wei et al.'s dataset *AMD* is `Trojan-SMS`, which comprises apps that silently send SMS messages to premium numbers owned by the malware authors. The third most common malware type is `Ransom`, also known as `Ransomware`, which generates monetary profit by encrypting the users' files and decrypting them only if users pay a ransom to the malware authors.

Manually analyzing a smaller dataset of 200 apps that we downloaded from the *AndroZoo* platform [10], we also found that the most common functionality adopted by malicious

apps is to either to display advertisements to the users or to fingerprint the devices on which they run (e.g., using methods such as `getDeviceID`, `getSimOperatorName`, or `getNetworkOperatorName`), prior to leaking such information to advertisement servers. In other words, the majority of malicious apps in this small dataset belonged to the `Adware` malware type. Furthermore, we found that the majority of servers contacted by such apps were renowned servers that are utilized by malicious and benign apps alike.

The wide adoption of `Adware` payloads is sensible for the following reasons. Firstly, the availability of different advertising APIs and libraries makes it straightforward to implement apps that display apps to the users even if they violate some aspects of user privacy [89]. Secondly, `Adware` is a malware type that researchers have disagreed vis-à-vis its malignancy, often referring to it as *grayware* or Potentially Unwanted Application (PUA) instead of malware [153, 15]. Some researchers even omit `Adware` from their studies altogether, such as Arp et al. in [15]. This confusion regarding the malignancy of `Adware` apps is likely to lead to some of those apps being labeled as benign and, perhaps, making it to app marketplaces. Lastly, unless the information leaked from a user's device directly violates a law, authors of `Adware` apps may evade the persecution of law enforcement authorities [137].

The payloads we discussed so far have a common characteristic, namely that they do not interfere with the functionalities of the benign code segments, which coincides with Li et al.'s findings that malicious functionalities are usually separated from their benign counterparts [75]. In other words, even if the code of malicious and benign functionalities is intertwined, as discussed earlier, the malicious functionalities do not alter the benign functionalities in the app (e.g., displaying the weather forecast). Fratantonio et al. [46] gave an example of a payload that delivers its malicious intentions by altering the original functionality of an app. They gave an example of a navigation app that returns a longer route between two points on a map to the user instead of the shortest or best possible one. In this case, the app continues to deliver its functionality (i.e., calculating a route between two points on a map), but not in an intended manner. In cases of emergency, this modification to the app's main, benign functionality can jeopardize the life of an individual in need of medical care. Technically, without identifying the trigger condition that instructs the app to return a longer route, this type of payload is challenging to detect solely by monitoring the app's runtime behavior.

2.4. Evasion Techniques

In order for their malicious apps to successfully pass the vetting process of app marketplaces and the scanning of antiviral software that might be used by Android users, malware authors employ different evasion techniques to conceal the malignancy of their instances [142]. Some of these evasion techniques alter the structure of the malicious apps' codebase, while others modify their runtime behavior to cope with static- and dynamic-based analysis. However, within the literature, they all fall under the umbrella of code *obfuscation*. Code

obfuscation encompasses techniques that render the source code of an app and/or its runtime behavior unintelligible to human beings [18]. Originally meant to protect sensitive code (e.g., license checking within commercial software), obfuscation has been widely adopted by malware authors to change the appearance of their instances and make it difficult for human analysts to understand their functionalities [134]. Furthermore, by changing the appearance of their apps, malware authors are able to evade detection by antiviral software, especially those that still rely on signature-based detection [67]. In this section, we discuss six techniques that are known to be frequently utilized by malware authors to write a malicious app that can evade detection, namely identifier renaming, string encryption, reflection, and Dynamic Code Loading (DCL), anti-analysis and disassembly, the usage of triggers and schedulers, and app repackaging.

2.4.1. Identifier Renaming

For the `Java` segment of Android apps, the most common technique to obfuscate source code is identifier renaming [38]. This technique alters the names of classes, methods, and variables from the human-understandable ones that developers use to keep track of and understand one another's code to ones that force the reverse engineer to analyze the source code in order to understand the functionality of each class, method, and variable. For example, the identifier names in Listing 2.3's original code indicate that the code belongs to a calculator class. However, after renaming the identifiers, the functionality of the code is blurred and not as obvious as before.

Listing 2.3.: Example of the obfuscation technique of identifier renaming.

```
1  // Original code
2  public final class Calculator extends AppCompatActivity{
3      private TextView inputScreen;
4      private String currentOperator = "";
5      private String display = "";
6      private String result = "";
7
8          public String performOperation(){...}
9  }
10
11 // Obfuscated with identifier renaming
12 public final class c extends u{
13         private TextView i;
14         private String j = "";
15         private String k = "";
16         private String l = "";
17
```

```
18          public String p(){...}
19  }
```

Albeit primitive, identifier renaming is a fast and effective method that is popular among android developers and malware authors [20]. The most renowned tool that implements this obfuscation technique is *ProGuard*, which is included in the Android Studio IDE [36]. More complex obfuscation techniques, including control-flow flattening, opaque predicates, and instruction set virtualization, can be applied to the `C/C++` portion of the Android app, given the availability of more tools that implement those techniques for the aforementioned language, for example, *Tigress* [117].

2.4.2. String Encryption

In addition to identifiers, string objects can also be obfuscated (e.g., using encryption [38]), to conceal their purpose and the overall purpose of the app. For Android malware, strings can reveal the Uniform Resource Locator (URL)'s contacted by the malicious app, the commands it receives from a remote service, or cryptographic keys, which are cues that can be used by antiviral software to detect those apps [153]. In some cases, entire code segments (e.g., classes), can be encrypted and stored as strings. These segments are then decrypted and dynamically loaded by the app during runtime. Wei et al. found that malware authors use different methods to encrypt or encode these strings (e.g., byte permutation, one-time pad encryption, and `Base64` encoding [153]). In the majority of cases, the encrypted strings are decrypted only during runtime. The keys used to decrypt those strings can be stored locally within the app's codebase or resource files or on a remote server to prevent it from being retrieved using static analysis.

2.4.3. Reflection and Dynamic Code Loading

The techniques of reflection and DCL were primarily meant to support the modularity and maintainability of `Java`-based apps, such as Android apps. Legitimate app developers usually rely on these two techniques within gaming apps to, for instance, support in-app purchases and the addition of new functionalities to apps. Malware authors, in contrast, use reflection and DCL to deploy their malicious payloads upon request [8, 75, 142]. So, instead of having the malicious code exposed, the code itself is shipped in a separate text file often concealed as a database file or a configuration file [107]. The malicious code will be loaded, parsed, and executed dynamically during runtime, effectively hindering its discovery using static analysis.

Listing 2.4.: Code snippet extracted from a malicious app belonging to the `Obad` malware family and shows the utilization of identifier renaming, string encryption, and reflection [153].

```
// Obfuscated code
ComponentName v0_3 = this.getComponentName();
Object[] v2_2 = new Object[3];
v2_2[2] = 1;
v2_2[1] = 2;
v2_2[0] = v0_3;
Context v1_1 = this.getAplicationContext();
Object v1_2 = Class.forName(IljIllj.IliijIIl(IljIllj.IliijIIl[104],
        IljIllj.IliijIIl[33]+1, IljIllj.IliijIIl[26])).
    getMethod(IljIllj.IliijIIl(IljIllj.IliijIIl[20],
    IljIllj.IliijIIl[20]|14, 183), null)
    .invoke(v1_1, null);

Class<?> v0_2 = Class.forName(IljIllj.IliijIIl(IljIllj.IliijIIl[107],
        IljIllj.IliijIIl[107]|24, 79));

int v2 = IljIllj.IliijIIl[138];
int v4 = IljIllj.IliijIIl[127]-1;
v0_2.getMethod(IljIllj.IliijIIl(v3, v4, v4 | 69),
        Class.forName(IlkIllj.IliijIIl(IljIllj.IliijIIl[7],
        IljIllj.IliijIIl[33]+1, 136)), Integer.TYPE, Integer.TYPE)
        .invoke(v1_2, v2_2);

// Deobfuscated code
ComponentName v0_3 = this.getComponentName();
Context v1_1 = this.getApplicationContext();
PackageManager v1_2 = v1_1.getPackageManager();
v1_2.getComponentEnabledSetting(v0_3,
        PackageManager.COMPONENT_ENABLED_STATE_DISABLED,
        PackageManager.DONT_KILL_APP);
```

As seen in Listing 2.4, the obfuscation techniques of identifier renaming, string encryption, and reflection are often combined by malware authors. In that figure, Wei et al. demonstrate how an instance of the malware family `Obad` utilizes the three techniques to make its code unintelligible to human analysts. Firstly, all identifiers were renamed to unintelligible strings, such as *IliijIII* and *IljIllj*. Secondly, method invocations were all made using reflection. Lastly, the names of methods to be invoked are encoded and stored in a byte array, called `IliijIIl`. The method `IljIllj.IliijIIl` retrieves the target method from such an array, decodes its name, and invokes it. The deobfuscated code shows that the obfuscated code uses the `PackageManager` to disable the current component (e.g., activity), without killing the app containing the component. This technique of enabling and

disabling components alters the runtime behavior of the app, which falls under a type of evasion technique that is meant to hinder the dynamic analysis of the app. We discuss this technique later in this section.

2.4.4. Anti-Analysis and Disassembly

Although they make it harder to understand the source code of malicious apps, the previous techniques still enable analysts to decompile and disassemble Android apps (i.e., into their `Smali` representations). Anti-analysis techniques attempt to obfuscate code by preventing analysts from seeing it altogether. Technically, anti-analysis techniques inject code that, in its compiled form, is difficult to recognize by disassemblers as valid instructions and are, hence, ignored [134]. One of the most effective techniques to prevent the effective disassembly or decompilation of Android apps is that of string encryption because encrypted code segments are not going to be recognized as valid `DEX` instructions by decompilers. Other techniques rely on creating bogus jump instructions. For example, in Listing 2.5, adding a one to the label `loc_401010` introduces what is known as a *rogue byte*, which tricks the disassembler into thinking that the jump location in the `jz` instruction lands at the byte `8B` and translate instructions starting from that point, which leads to a `call` instructions that calls a bogus address `8B4C55A0h`. This type of technique is usually employed on the `C/C++` level in Android apps to protect important assets to the malware author, such as encryption keys or addresses to remote control servers.

Listing 2.5.: Example of a jump instruction to a location with a constant value.

```
1  ;---------------
2  ; Original code
3  ;---------------
4  74 01       jz    short near ptr loc_401010
5  E8          db 0E8h
6              loc_401010+1:
7  8B 45 0C    mov eax, [ebp+0Ch]
8  8B 48 04    mov eax, [eax+4]
9  0F BE 11    movsx edx, byte ptr [ecx]
10
11 ;------------------
12 ; Manipulated code
13 ;------------------
14 74 01              jz    short near ptr loc_401010+1
15                    loc_401010+1:
16 E8 8B 45 0C 8B     call near ptr 8B4C55A0h
17 48                 dec eax
18 04 0F              add al, 0Fh
```

```
19  BE 11 83 FA 70    mov esi, 70FA8311h
```

2.4.5. Triggers and Schedulers

The obfuscation techniques we discussed earlier are meant to hinder the static analysis of apps. To hinder dynamic analysis as well, malware authors design their apps to conceal malicious behaviors within benign ones by delaying or scheduling their execution. Consider, for example, a malware analyst that examines the runtime behavior of Android apps in the format of the sequences of the API calls they issue during runtime. Such an analyst is in pursuit of API calls and their arguments that indicate malicious intentions. If the malicious code segments are protected via triggers and schedulers, it is likely that the API calls examined by the analyst will often lack traces of malignancy.

The notion of using triggers to delay and control the execution of malicious payloads is not unique to Android malware. In fact, Windows-based malware has been utilizing triggers for decades [134]. Triggers are usually implemented as conditional statements that control the execution of the malicious code. The conditions that such statements employ are usually not random and coincide with the malware authors' vision of how the malicious app should function. For instance, some malware authors design their malicious apps to execute their payloads on certain dates and times [30], in specific locations [157], based on some secret or password remotely sent from the author (i.e., activation code) [133], or if the environment on which they run satisfies some condition (e.g., not a virtual environment) [134]. Android malware utilizes the same concepts to implement what we refer to as *time-based*, *location-based*, *secret-based*, and *system-based* triggers.

Time-based Triggers As the name implies, time-based triggers execute the malicious payloads on a specific date and time. This type of trigger has been historically referred to as *time bombs* [46, 106]. Listing 2.6 illustrates three time-based triggers that execute malicious code whenever the current system time and date equal values pre-set by the malware authors. Malware authors can complicate those conditions using different techniques. For example, `DateTime` objects can be directly compared instead of formatted dates, the conditions can be nested, or less intelligible timestamps (e.g., Epoch timestamps), can also be utilized.

Listing 2.6.: Examples of *time-based* triggers.

```
1  import java.util.Calendar
2  Calendar calendar = Calendar.getInstance();
3  String currentTime = new SimpleDateFormat("HH:mm",
   ↪  Locale.getDefault()).format(new Date());
4  String currentDate = new SimpleDateFormat("dd-MM-yyyy",
   ↪  Locale.getDefault()).format(new Date());
```

```
String currentDateTime = new SimpleDateFormat("dd-MM HH:mm",
 ↪  Locale.getDefault()).format(new Date());
// Execute code everyday at 14:00
if (currentTime.equals("14:00"))
        // execute malicious code
// Execute on April 1st, 2020
if (currentDate.equals("01-04-2020"))
        // execute malicious code
// Execute every November 5th at 17:00
if (currentDateTime.equals("05-11 17:00"))
        // execute malicious code
```

Location-based Triggers Location-based triggers rely on the location of the device on which the malicious app is running. This location can sometimes be relevant to the app's functionalities. For example, researchers at `Kaspersky` discovered an Android malware, designated `ViceLeaker`, that is designed to steal sensitive information from devices of users in Middle Eastern countries [48]. So, it makes sense for such malicious apps to check the location of their devices prior to executing their malicious payloads tailored for specific types of users. The most common technique to retrieve the location of an Android device is by using its GPS module and location services. The longitude and latitude returned by the GPS module can be too abstract or might require additional processing to fingerprint the device's country. Consequently, malware authors usually infer the device's location using the name of the network carrier the device is registered to, the country the device's Subscriber Identification Module (SIM) card is registered to, or perhaps the device's Internet Protocol (IP) address [4]. Using those API calls are also more subtle than using the location services, especially since they do not require to be granted permissions. Listing 2.7 illustrates some techniques to fingerprint the device's location and use it as a trigger.

Listing 2.7.: Examples of *location-based* triggers.

```
// Get instance of the device's telephony manager
TelephonyManager manager = (TelephonyManager)
 ↪  getApplicationContext().getSystemService(Context.TELEPHONY_SERVICE);
String countryCode = manager.getNetworkCountryIso();
String carrierName = manager.getNetworkOperatorName();
String mPhoneNumber = manager.getLine1Number();
// Execute if the carrier's country is the United States
if (countryCode.equals("us"))
        // execute malicious code
// Execute if the carrier's name is XYZ
if (carrierName.equals("XYZ"))
```

```
11          // execute malicious code
12  // Execute if the phone number has a German prefix
13  if (mPhoneNumber.startsWith("+49"))
14          // execute malicious code
```

Secret-based Triggers This type of trigger relies on secret values or passwords provided from outside entities to execute malicious code. Typically, secret-based triggers rely on values that are stored locally within the malicious app as primitive types (e.g., integers, strings, floats, etc.). In Android apps, such values can either be stored within the app's code, in resource files, or external files. The other option is to receive the secret values from a remote source (e.g., via SMS messages [153, 108]). Malware authors usually design their instances so that the values they receive will execute certain code segments in accordance with the authors' intentions. Effectively, the secret values are commands that execute certain functionalities in malicious apps. This model is usually employed either by botnet controllers or by authors of `Ransom` apps [153]. In Listing 2.8, we give examples of a number of methods Android malware can read secret-values, namely from within the app's premises or by receiving them from remote sources.

Listing 2.8.: Examples of *secret-based* triggers.

```
1   import java.io.*;
2   String secretValue = "mysecret";
3   // Get input from EditText
4   EditText editText = (EditText) findViewById(R.id.EditText1);
5   String userInput = editText.getText().toString();
6   if (userInput.equals(secretValue))
7           // execute malicious code
8   // Get string from strings.xml resource file
9   String resInput = getResources().getString(R.string.String1);
10  if (userInput.equals(resInput))
11      // execute malicious code
12  // Get string from received SMS
13  public void onReceive(Context context, Intent intent){
14      if (intent.getAction().equals(Telephony.Sms.Intents.SMS_RECEIVED_ACTION)) {
15              String smsSender = "";
16                  String smsBody = "";
17                  for (SmsMessage smsMessage :
                    ↪ Telephony.Sms.Intents.getMessagesFromIntent(intent)) {
18                   smsSender = smsMessage.getDisplayOriginatingAddress();
19                          smsBody += smsMessage.getMessageBody();
20                          if (smsSender.equals("[AttackerNumber]")
```

```
                                if (smsBody.contains("LeakContacts"))
                                    // leak user contacts
                                else if (smsBody.contains("EncryptDevice"))
                                    // encrypt SD card contents
                                ...
                    }
            }
        ...
}
// Receive commands from network
serverSocket = new ServerSocket(8080);
socket = serverSocket.accept();
output = new PrintWriter(socket.getOutputStream());
input = new BufferedReader(new InputStreamReader(socket.getInputStream()));
// In a thread
while (true){
    String message = input.readLine();
    if (message.contains("execute"))
        // execute malicious code
}
```

System-based Triggers The name system-based can be misleading given that other types of triggers rely on values returned by the system as well. For example, time-based triggers rely on values returned from the Android system as responses to API calls made by apps. By system-based, however, we refer to system software and hardware properties, such as the Operating System (OS) version running on the device, the device's IMEI, or its Media Access Control (MAC) address. Similar to benign apps, some system properties are checked to ensure that the malicious apps run within the environments they are meant to run (e.g., a particular Android OS version). Other system properties are checked as part of a method to hinder dynamic analysis of malicious apps; malware authors utilize some system-based triggers that check whether an app is running on a virtual environment (e.g., Android emulator). Such virtual environments usually leave traces within the system that can be easily checked by a malicious app. For example, Android emulators usually use a default IMEI value of 15 zeros. This can be utilized, as seen in Listing 2.9, to infer whether a device is virtual and, in turn, whether to withhold the execution of malicious code segments.

Listing 2.9.: Examples of *system-based* triggers.

```
import java.util.*;
// Check the SDK version of the current system. If more recent than Android 4.4.
↪ KitKat
```

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT)
    // execute malicious code
// Check whether device is virtual
// Get instance of the device's telephony manager
TelephonyManager manager = (TelephonyManager)
 ↪  getApplicationContext().getSystemService(Context.TELEPHONY_SERVICE);
String imeiNumber = manager.getImei();
// Get instance of the device's WiFi manager
WifiManager managerW = (WifiManager) getSystemService(Context.WIFI_SERVICE);
WifiInfo info = managerW.getConnectionInfo();
String macAddress = info.getMacAddress();
if (macAddress.length > 0 && !imeiNumber.equals("000000000000000"))
        // execute malicious code
```

Logic-based Triggers The last type of triggers we discuss is one that depends on the app's logic rather than values drawn from the underlying system. For example, as seen in Listing 2.10, a malware author may elect to execute malicious code injected into a calculator app only if the result of the arithmetic operation performed by the user is 50. It is expected that such triggers merge with the functionality of malicious apps to evade detection. That is to say, a calculator app should use a logic-based trigger related to arithmetic operations, whereas a gaming app can trigger malicious payloads upon, for instance, scoring certain high scores. So, the conditions used by a logic-based trigger hinge on the app itself and is up to the creativity of the malware author. Logic-based triggers are similar to secret-based triggers in terms of relying on secret values to execute malicious code. However, the secrets on which logic-based triggers rely are intertwined with the app's functionality and do not rely on values from outside entities.

Listing 2.10.: Examples of *logic-based* triggers.

```
import javax.script.ScriptEngineManager;
import javax.script.ScriptEngine;
// Retrieve arithmetic operation as string
EditText editText = (EditText) findViewById(R.id.EditText1);
String userInput = editText.getText().toString();
ScriptEngineManager mgr = new ScriptEngineManager();
ScriptEngine engine = mgr.getEngineByName("JavaScript");
// Use Javascript engine to evaluate input as an arithmetic operation
if (engine.eval(userInput) == 50)
    // execute malicious code
```

If-statement-based triggers are not the only way a malware author can intentionally delay

the execution of malicious code. As discussed earlier, Android apps have multiple entry points, one of which is via broadcast receivers. Malware authors sometimes implement their apps to trigger malicious code upon receiving system notifications (e.g., `android.intent.action.BOOT_COMPLETED`). In Listing 2.8, we give an example of receiving secret values using an SMS message. The code segment checking this secret or at least retrieving the body of the received text message will only be triggered if the malicious app registers to system notifications of new incoming SMS messages. In other words, if-statements, along with system notifications, can be combined to form more complex triggers, as seen in Listing 2.11. The malicious payload in this code segment executes upon receiving a new SMS message; if the message body starts with the string *"ak401"*, the app checks whether its own codebase has been modified prior to decrypting a file containing a list of mobile operators the app targets. If the mobile operator of the current device matches one of the target operators (i.e., a system-based trigger), the malicious app dynamically loads the malicious code from a file called `anserver.db`.

Listing 2.11.: Combining different types of triggers to hide the apps's malicious intentions [107].

```
String hashValue = "389d90db090f0f0303030030d98676ie03";
// called when an SMS message arrives
public void onReceive(Context c, Intent intent){
    String smsMessage = intent.getMessageBody();
    // checks content of SMS message
    if (smsMessage.startsWith("ak401"){
        String certificateHash = this.getCertificate().toString();
        // integrity check whether the APK got modified
        if(certificateHash.equals(hashValue)){
            // get MCC and MNC codes
            String mobileOperator = getNetworkOperator();
            File encryptedFile = readFileFromStorage();
            File decryptedFile = decryptFile(encryptedFile);
            boolean containsMobileOp = false;
            Reader bf = new Reader(decryptedFile);
            String line;
            // checks whether file contains mobile operator
            while ((line = br.readLine()) != null){
                if (line.equals(mobileOperator)) {
                    containsMobileOp = true;
                    break;
                }
            }
            // targeted attack against specific network
```

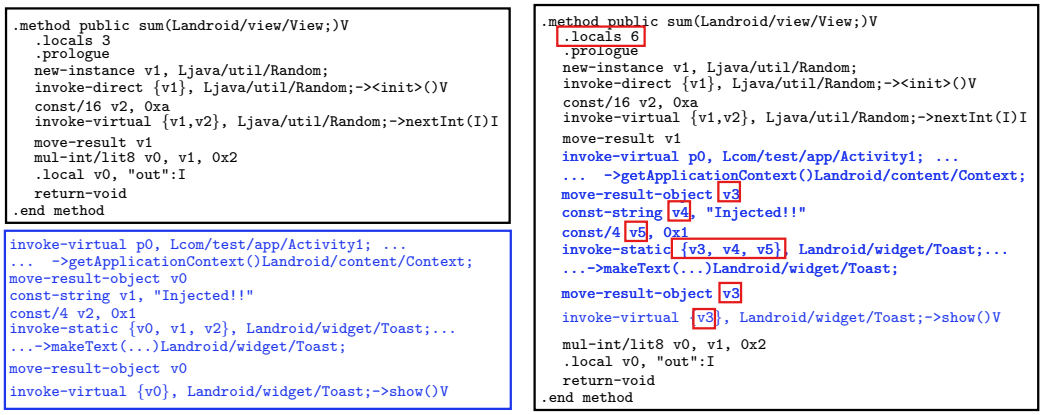
```
25              if (containsMobileOp){
26                  // dynamic class loading, expects a dex file,
27                  // even though file suffix is .db
28                  DexClassLoader dcl = new DexClassLoader("anserverb.db");
29                  Class clazz = dcl.loadClass("BaseABroadcastReceiver");
30                  Method method = clazz.getMethod("onStart", Intent.class);
31                  boolean returnValue = (boolean) method.invoke(intent);
32                  if (returnValue == false){
33                      // target location : aborts delivery message
34                      this.abortBroadcast();
35                   }
36               }
37          }
38      }
39  }
```

Apart from some time-based triggers, the triggers listed above do control whether a malicious code segment will execute, but do not control how often it will execute. For example, the malicious code in Listing 2.6 may never execute, if the app is launched on times and dates that do not match the triggers' conditions. To increase the likelihood of malicious code execution yet maintain stealthiness, malware authors usually rely on the techniques of *scheduling* [153]. In addition to using conditional statements to trigger malicious payloads, malware authors can force the frequent execution of particular code segments. The most common scheduling technique relies on the `Timer` class [37, 153], which schedules tasks for future execution in a background thread. So, instead of waiting for users to start apps and check whether the current system time matches a time-based trigger, a malware author can include this code in a service that starts upon the completion of a device boot and checks for the condition every hour, for instance.

2.4.6. App Repackaging

The last evasion technique we discuss is that of repackaging. Unlike standalone malware, with repackaging, Android malware authors attempt to obfuscate their malicious payloads by grafting them into pre-existing benign apps. The exact format of those benign apps seems to be disputed [119]. On the one hand, benign apps can be open source apps (e.g., *K9 Mail* [3]), which facilitates the repackaging process as malware authors need to write their payloads in high-level languages such as `Java` and `C/C++`. On the other hand, benign apps can take the form of pre-compiled Android apps in the format of `APK` archives. In this case, malware authors attempt to disassemble or decompile the code in the `classes.dex` file and the shared object libraries, in order to be able to add their malicious payloads. The first approach is straightforward and does not require further explanation; so, we focus on the second approach of repackaging.

Figure 2.4.: A demonstration of the Smali format and the ease of adding new functionality to existing code. The code in black is the original Smali code we target, and the code in blue is the Smali code we wish to inject in the original code. The code snippet on the right shows the merged Smali code after updating the number of variables to accommodate the newly-inject code [118].



Repackaging an Android app with new functionality can be reduced to the process of manually merging a chunk of code depicting the new functionality (hereafter rider code) with that of the app's original code while maintaining the syntactic and semantic integrity of the app's original code. Ideally, such code segments should be in a high-level language, which requires decompiling the app's code. There are tools, such as Jadx [135], that decompile DEX Bytecode into Java. However, the process of recompiling the app after adding the malicious payloads might not be as straightforward [34]. A more stable approach is to operate on the intermediate level of Smali.

So, repackaging a benign Android app with malicious payloads can, in turn, be reduced to the process of merging a malicious Smali code within the disassembled Smali code of the benign app. We use Figure 2.4 to illustrate such process. In this figure, the malware author attempts to merge the original code (top left) that multiplies a random integer by two and stores the output in the variable out, with rider code (bottom left) that displays a Toast dialog with the string "*Injected!!*".

The location to inject the rider code is up to the malware author. In this example, we assume that the malware author opted to inject the rider code right after the generation of the random integer, and before multiplying it by two. The location chosen to inject the rider code dictates some minor changes to both the original and rider code snippets to maintain the stability of the app. For instance, the original code utilizes only three registers as variables (i.e., v0, v1, and v2), and declares their count using the statement .locals 3. Since the rider code utilizes the exact same registers, merging both code snippets without any modification of the registers risks undermining the app's stability. One solution is to update the register numbers used by the rider code to v3, v4, and v5, and updating the

number of registers utilized by the method to six via the `.locals 6` statement.

To avoid this confusing situation, an attacker might opt to inject the rider code either at the very beginning or the very end of a method. This approach circumvents the necessity to modify register counts, as it separates the functionalities of both the original and rider code. In other words, the registers used by one code snippet will not be further needed after the completion of its functionality and can, hence, be safely overwritten. After merging the code, attackers can re-assemble the new `Smali` code, sign the resulting APK archive with their own private keys, and release the new app to their marketplaces of choice. Attackers usually alter some attributes of the original app (e.g., name, colors, fonts, etc.), to sell the repackaged app as a new version of the original one.

The previously-described process is constant with a few special cases. Consequently, it can be automated. Li et al. found that piggybacking benign apps with malicious payloads is mostly automated with the payloads being usually reused across several piggybacked apps [75]. This facilitates the mass production of Android malware, making the job of malware analysts even harder to cope with the large amounts of Android malware existing in the wild. To shed light on this problem, researchers have implemented tools, such as *Repackman* [118] and *Packadroid* [1], to automatically repackage Android apps with malicious payloads.

2.5. Detection Rates of Android Malware Types

In the previous sections, we differentiated between different types of Android malware based on the functionalities their payloads deliver. We also discussed that some of these types might be perceived as less malicious than others based on the damage they may inflict on Android devices and their users and the legal implications they pose to the authors of such malware types. In this section, we explore whether different Android malware types draw different detection rates[2] from `VirusTotal` scanners, and attempt to explain the reasons behind such differences in detection rates, if any. To that end, we used apps in the *AMD* dataset because we know their malware families and types. We grouped those apps by the malware types they belong to (i.e., `Adware`, `Ransom`, `Trojan`, etc.), and for each app in each group, calculated its detection rate at different points in time between November 2018 and November 8^{th}, 2019. We calculate this rate instead of solely considering the total number of scanners because the total number of scanners included in a `VirusTotal` scan report differs from one app to another. Lastly, we averaged the detection rates within every group at every point in time and plotted them in Figure 2.5. We also tabulate the average detection rates achieved on November 8^{th}, 2019 in Table 2.1.

The results in Table 2.1 show that detection rates indeed differ from one malware type to another. For example, we found that apps that belong to the `Adware` type (emboldened green) have lower detection rates than other malware types, such as `Ransom` (emphasized red) and `HackerTool` (underlined orange). Further, we found that, on average, 51.5% of

[2]We define the detection rate of a malicious app as the percentage of `VirusTotal` scanners that managed to deem it as malicious.

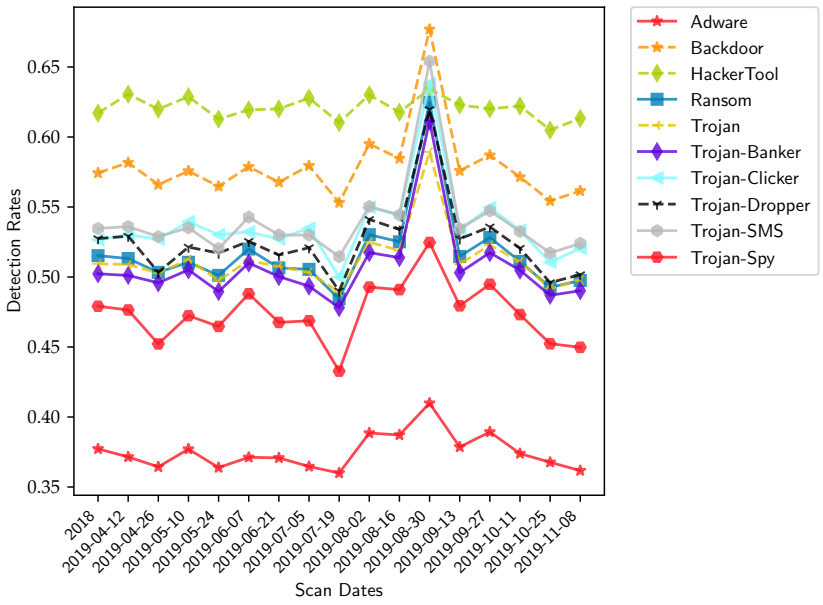


Figure 2.5.: The mean detection rates of `VirusTotal` scanners against different malware types found in the *AMD* dataset between November 2018 and November 8^{th}, 2019. We found that the spike in performance on August 30^{th}, 2019 is due to an increase in the number of scanners used by `VirusTotal` to scan the apps in the *AMD* dataset, which correctly labeled the apps as malicious. In Chapter 4, we demonstrate and discuss `VirusTotal`'s manipulation of the number and versions of scanners it includes in the scan reports of apps.

Table 2.1.: The mean, median, and standard deviation of detection rates recorded by **all** VirusTotal scanners on apps in the *AMD* dataset as of November 8^{th}, 2019, grouped by malware type and sorted by the percentage of apps in the dataset.

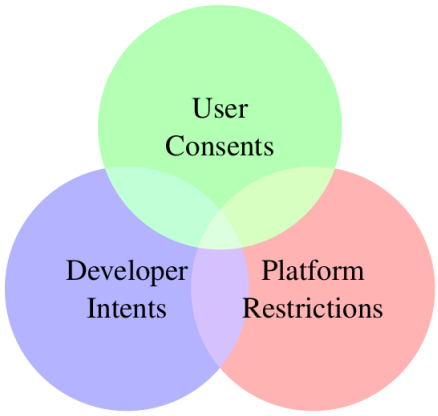
Metrics \ Malware Types	Adware	Trojan-SMS	Ransom	Trojan-Spy	Backdoor	Trojan-Banker	Trojan	HackerTool	Trojan-Dropper	Trojan-Clicker
Total Apps per Type (%)	14163 (57.7%)	3219 (13.1%)	2148 (8.74%)	1851 (7.5%)	1047 (4.26%)	929 (3.78%)	517 (2.1%)	347 (1.41%)	301 (1.2%)	31 (0.1%)
Mean Detection Rate	**0.36**	0.52	*0.49*	0.45	0.56	0.49	0.50	0.61	0.50	0.52
Median Detection Rate	**0.35**	0.52	*0.50*	0.46	0.57	0.49	0.51	0.65	0.50	0.51
Standard Deviation Detection Rate	0.05	0.05	0.02	0.03	0.07	0.03	0.06	0.10	0.05	0.05

VirusTotal scanners fail to recognize the malignancy of apps in the *AMD* dataset, which implies that some threshold-based labeling strategies that label apps as malicious if 50% of the VirusTotal scanners deem them as such [153] might fail to recognize that an app is malicious. One possible explanation for such discrepancies in the detection rates among malware types is that the detection rate of apps in a particular malware type is correlated with the damage it may inflict on the infected device. However, to check whether this correlation exists, we first need to rank different Android malware types according to the damage they inflict, which seems **impossible** to do. First, we cannot assert, for instance, that the damage inflicted by all Ransom apps is more than that inflicted by all Trojan apps. Second, inflicted damage is subjective, and its definition might differ from one individual to another depending on whether they consider the affected resources (e.g., local files stored on a device), as important.

The question that follows is: What causes such discrepancies between the detection rates of different malware types? Antiviral software companies tend not to share the methods they apply to analyze and label apps [67, 99]. Without access to such information, it is **not possible** to conclusively pinpoint the reasons behind the disagreements among scanners. That is, any answer we give to this question is **a matter of speculation** and, hence, debatable.

As mentioned in Chapter 1, the typical process of malware analysis and detection that is presumed to be adopted by anvitiral software companies often involves human analysts that give the final decision on whether to label an app as malicious or benign. Those analysts might have different views on what makes an app malicious, which makes the labeling process subjective. However, we assume that large antiviral software companies adopt decision processes to deem apps as malicious and benign rather than delegate the final decision to individual analysts. In [54], Hurier argued that "the decision to classify

Figure 2.6.: The definition of malware at the intersection of three other notions of user consents, developer intents, and platform restrictions according to Hurier in [54]



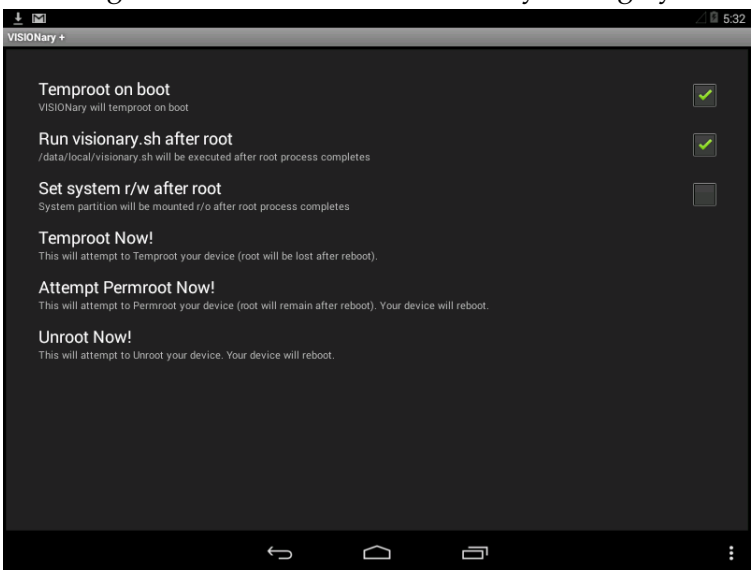
an application as malicious relies on an implicit contract between end-users, developers, and digital marketplaces", as seen in Figure 2.6. Apps that fail to satisfy this contract (e.g., because the end-user was not informed about the intents of the developer or it violated some platform restrictions), should be labeled as potentially malicious. The goal of the security experts, in this case, is to report apps that violate such a contract.

Unfortunately, this model of malignancy has two shortcomings. Firstly, not all apps that fail to clearly state their intentions should be considered suspicious. Moreover, some apps cannot predict their future behavior beforehand because they are customizable (e.g., via in-app purchases). Secondly, this model does not capture some of the results we tabulated in Table 2.1. For example, despite making their intentions clear (i.e., rooting the device as seen in Figure 2.7), and not violating any platform restrictions, apps in the `Lotoor` family were labeled as malicious by many `VirusTotal` scanners.

We devised a model that might better explain the results above, particularly the disagreements among antiviral scanners vis-à-vis the labels of Android apps. This model is based on the assumption that labeling an Android app as malicious is a matter of the perspective(s) assumed by whoever is labeling the app. In this model, there are three categories of perspectives that can be considered upon labeling an app and impact its label, namely the **user** perspective, the **supplier** perspective, and the **security researcher** perspective.

The majority of users do not possess the technical knowledge necessary to deem apps as malicious or benign. However, if the functionality of an app is continuously interrupted (e.g., via aggressive advertisements), an app intrudes on their personal lives (e.g., by leaking intimate photographs), an app denies them access to their data (e.g., contacts), or causes them financial loss (e.g., via sending SMS messages to premium numbers [108]) they will indeed deem it as malicious. Any other malicious functionality that is unnoticeable, such as

Figure 2.7.: A screenshot of the app (`959c804a1b621703c68196cfc243db8fc7300e4c`), which belongs to the `Lotoor` malware family during dynamic analysis.

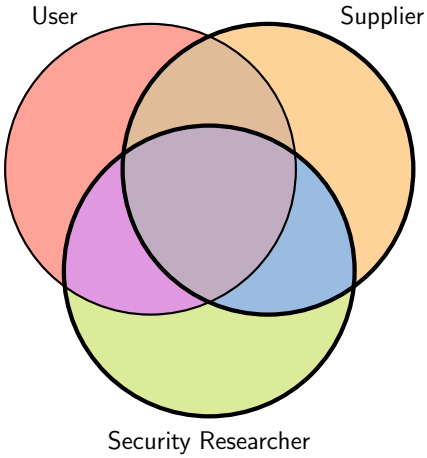


leaking the device's location or fingerprinting the device for targeted advertisements [19], might pass as benign for users.

As for suppliers (i.e., app developers, device manufacturers, and marketplace maintainers), their primary objective is to make a profit. In the case of Android, the majority of profit is generated via advertisements, as mentioned earlier. Activities that intentionally hinder profit generation are, therefore, considered undesirable. This includes any activities that discourage users from using the Android platform because it prevents the suppliers from realizing their objectives. In this aspect, the users' and supplier's interests are intertwined. However, some activities are malicious, such as fingerprinting user devices without their consent for targeted advertisements, which are often tolerated by vendors because they contribute to generating profits. According to this line of argumentation, the supplier's perspective is that of a compromise between user satisfaction and loyalty and monetary profit. In other words, any activities that do not prevent users from purchasing Android devices and acquiring apps from marketplaces whilst abiding by general policies are considered benign.

The security researcher's perspective is a neutral one that deems apps as malicious if they threaten the confidentiality, integrity, or availability of the device or the data it stores. We differentiate between a security researcher and a malware analyst in that the former does not adhere to a particular policy or decision process that is enforced by a particular antiviral software firm. To give an example of this perspective, consider the app `xxxaa.wjysq.com`, which we manually analyzed. We found that the app frequently shares the current longitude and latitude of the device with the domain `yota365.com`, which belongs to the people's

Figure 2.8.: The perspectives that can be adopted upon labeling (Android) apps as malicious and benign.



congress in China. Given that tracking users without their consent has recently been criminalized in China [70]; this makes the app malicious from ethical, legal, and technical perspectives. So, regardless of the user and supplier perspectives, a security researcher is expected to deem the app as malicious.

Using this model, we can hypothesize that the disagreements between different antiviral scanners regarding the labels of apps as follows. The verdicts given by antiviral scanners depend on which individual perspective or intersection of perspectives the antiviral firm maintaining the scanner adopts. For example, if an antiviral company solely adopts the user perspective, then it might have a strict policy that `Adware` apps are malicious. Unlike Hurier's model, our model can justify why more than 60% of `VirusTotal` scanners agree upon the malignancy of `HackerTool` apps, namely that they happen to adopt the same perspective(s). Unfortunately, as mentioned above, without access to the decision processes adopted by antiviral firms to label apps, no model can actually explain why different scanners dis/agree upon the labels of apps.

2.6. Summary

Despite being relatively easy to implement, Android apps are complex in structure. This complexity allows malware authors to write sophisticated malicious apps that deliver a multitude of functionalities and implement different evasion techniques, such as obfuscation,

dynamic code loading, triggers, and scheduling, which might complicate the process of analyzing and labeling Android apps. Labeling apps is further complicated by the fluidity of defining what is malicious. In this chapter, we found that `VirusTotal` scanners disagree upon the labels given to the Android apps, with almost 50% of the scanners on average mislabeling malicious apps in the *AMD* dataset as benign. This insight will be corroborated in Chapter 4 that threshold-based labeling strategies that need relatively high percentages of `VirusTotal` scanners to deem apps malicious in order to labels apps as such (e.g., 50% of scanners), would be unable to accurately label malicious apps. Furthermore, we found that `VirusTotal` scanners agree upon the malignancy of certain malware types more than others, which is a concern of **(RQ2)**.

3. Android Malware Detection

This chapter provides an overview of Android malware detection, its theoretical limitations, and the objectives it pursues in practice. In this chapter, we focus on ML-based malware detection methods and enumerate the challenges hindering effective detection. Parts of this chapter have previously appeared in peer-reviewed publications [120] and [119], co-authored by the author of this thesis.

Android malware has been disrupting services, inflicting a monetary loss on individuals and corporations alike, and undermining trust in the safety of apps and systems, instigating researchers to devise methods to detect it [69, 115, 32, 16, 57]. Typically, the detection process includes elements of human expertise. For example, if automated methods fail to detect a malicious app, a human analyst might be consulted to decide upon the app's nature. To cope with the increasing number of malicious apps, the effectiveness and applicability of fully automated detection methods are researched. However, automated detection of (Android) malware suffers from theoretical (Section 3.1) and practical (Section 3.2) limitations that researchers aspire to address or circumvent. In this chapter, we discuss those limitations that face different approaches to automated malware detection and give an example of each approach in Section 3.3. Courtesy of its popularity within the academic research community, in this thesis, we focus on ML-based detection methods as a case study of automated malware detection and further discuss its components (Section 3.4) and the issues that hinder it from being more effective against (Android) malware (Section 3.5).

3.1. Malware Detection in Theory

In malware research, the earliest definition of malicious software is usually accredited to Cohen, who discussed the notion of *computer viruses* as programs that–akin to the notion of biological viruses and inspired by John von Neumann's definition of self-replicating automata [149]–copy their malicious functionalities to benign programs, so that they can execute every time the infected benign programs are launched [29, 28]. Along with this definition, Cohen provided proof that shows that the existence of an automated detection method that detects computer viruses and, by extension, malicious software is contradictory [28].

Cohen's proof is based on a contradictory example of a computer virus (`CV`), seen in Listing 3.1, that infects a target executable if and only if it is **not** detected by a detection method (`D`) as a virus. The statement that this virus contradicts is that there exists a detection

method (D), which is essentially a program that takes in an arbitrary program (P) as an input and outputs a decision whether (P) is a virus based on its appearance (e.g., call graph, source code, or control flow graph). It follows that D(CV) should output a value labeling (CV) as a virus. However, according to (CV)'s definition in Listing 3.1, the program will execute its malicious functionality of infecting other executables and *do-damage* (i.e., act as a computer virus), if and only if it is **not** detected by (D). Being detected by (D), (CV) will not execute its functionality, effectively not acting as a computer virus. So, the definition of (CV) creates a paradox in which (CV) is deemed as a computer virus, yet does not act like one.

Listing 3.1.: Fred Cohen's Contradiction of the Decidability of a Virus (CV) [28].

```
program contradictory-virus:=
{...
    main-program:=
        {if ~D(contradictory-virus) then
            {infect-executable;
             if trigger-pulled then do-damage;
            }
         goto next;
        }
}
```

The proof above does indeed generalize to any malicious software, including Android malware. In Chapter 2, we discussed the notion of triggers and schedulers that delay the execution of malicious functionalities within a malicious Android app. Triggers and schedulers do not need to be as complex as the one in Listing 3.1; any trigger that prevents a malicious app from acting malicious (e.g., temporal-based trigger), would fit the description above, which renders the existence of any automated detection method that always recognizes the malignancy of an app impossible.

In addition to this proof, Adleman related the problem of automatically detecting malware to *Rice's theorem* [109] and the *halting problem* [146], to show that the problem of malware detection is undecidable [65, 5]. Recall that Rice's theorem states that if a program (P) that can determine whether another program (P′) has a functional property (x) exists, then the halting problem can be decided, which is indeed not possible. A property of a program (P′) is called a functional property if (a) it describes input-output (IO) behavior (i.e., how P′s inputs and outputs are related), and (b) it need not be possessed by all programs. In terms of Android malicious apps, for example, a functional property can be that an app encrypts the user's files upon receiving instructions (i.e., inputs) from a remote server and returns a confirmation to such a server (i.e., output).

Consider the program (P) to depict an automated detection method (e.g., a ML-based detection method), the program (P′) to depict a malicious Android app, and the property

(x) to be, for example, the execution of malicious code after the realization of a conditional statement. According to Adleman's proof, detecting the malignancy by of (P′) by unveiling that it contains the malicious property (x) is undecidable.

3.2. Malware Detection in Practice

The proofs in the previous section imply the impossibility of having an automated malware detection method that can detect all types of malware. However, the examples given in those proofs do not reflect the reality of (Android) malware. Firstly, while some antiviral software attempt to label apps as malicious according to their behaviors, the majority of them rely on examining apps statically. So, a malicious app does not have to exhibit malicious behaviors during runtime in order to be labeled as malicious. In fact, it is considered an advantage to detect the malignancy of an app before it executes on the end user's device. This is the main reason behind antiviral software statically basing their judgment of an app's malignancy primarily on its structure and codebase. Secondly, antiviral software is never designed for discerning the halting problem before deciding upon the malignancy of an app.

The practical problem facing automated malware detection is that malicious apps continuously evolve. Malware authors can adapt to their malicious apps being detected by antiviral software by modifying or entirely re-writing them [67]. Furthermore, malware authors adopt new Tactics, Techniques, and Procedures (TTP) that are more relevant to the current technologies used by end-users and the vulnerabilities that can be exploited to infect their devices. As discussed in Chapter 1, conventional semi-automatic malware detection methods cannot cope with this continuing evolution. In this context, researchers have developed a plethora of automated detection methods for different types of malware, including Android malware, that attempt to mitigate the limitations of their conventional counterparts [115, 74, 142, 145, 131, 132, 79]. In the following section, we categorize these malware detection methods with the focus on Android malware and ML-based detection methods.

3.3. Malware Detection Methods

In Section 1.1, we defined the problem of malware detection as that of matching a never-seen-before app (i.e., test app), to one or more of the previously-seen malicious or benign apps stored in a repository. The main differences between Android detection methods dwell in (a) how they represent and store apps, and (b) how they match test apps to the ones stored in the repository. There are a multitude of ways to represent Android apps that includes app call graphs [47], vectors of numerical features extracted from the apps' APK archives [119, 32, 145], traces of the API calls issued by apps during runtime [120, 87], etc. These representations can be cateogrized into *static* and *dynamic* representations of an app. Static representations are extracted from apps before being executed. The primary source of static features is an app's codebase found in the `classes.dex` file along with metadata

Table 3.1.: A categorization of Android malware detection methods.

Static Heuristic-Based (e.g., *Codebase Similarity and Compiler fingerprinting*)	Dynamic Heuristic-Based (e.g., *Sequence Alignment*)
Static ML-Based (e.g., *Classification with Linear SVM*)	Dynamic ML-Based (e.g., *Anomaly Detection with HMM*)

found in the `AndroidManifest.xml` file. Dynamic representations are extracted from or depict an app's runtime behavior (e.g., the sequence of API calls issued by an app during runtime).

As for the techniques utilized by detection methods, they can also be categorized into two categories, viz. *heuristic-based* and *Artificial Intelligence (AI)-based*. Heuristic-based detection methods are implemented to reflect the domain knowledge that malware analysts and researchers possess. Such domain knowledge is usually implemented as simple heuristics that check for particular patterns encountered by human experts during the analysis of malware. For example, Stazzere [136] suggests using compiler fingerprinting to detect Android repackaged malware based on the following assumption: legitimate developers usually have access to their apps' source code that they modify within IDEs and, hence, their apps should be compiled using the `dx` or `dexmerge` compilers that ship with the Android SDK. Consequently, apps that are compiled using third-party compilers used by reverse engineering tools (e.g., `dexlib`), should raise suspicions.

AI-based detection methods usually rely on machine learning algorithms to match apps to those in the detection method's repository. Since they mostly rely on machine learning algorithms, we refer to AI-based detection methods as ML-based detection methods. In addition to speed and scalability [100, 91], machine learning algorithms can find patterns that help make decisions about apps' natures that are not obvious to human researchers. For example, feature selection algorithms can indicate that malicious apps tend to utilize more permissions than their benign counterparts [75, 161], which is not easy to spot by humans.

Combining the categories of app representation and matching techniques leads to four categories of Android malware detection methods, seen in Table 3.1. In reality, those four categories are often combined to come up, for example, with hybrid app representations. For example, Miller et al. implemented a detection method that incorporates expert decisions about the labels (i.e., malicious or benign), of app representations into the process of training a machine learning classifier meant to classify Android apps as malicious and benign [91]. In this thesis, we focus on static, ML-based methods to detect Android malware as our case study. In the following sections, we provide background knowledge about the typical process of acquiring and labeling Android apps using `VirusTotal`, statically extracting and selecting features from the APK archives of such apps, and using the extracted features

to train and validate different types of ML algorithms to detect Android malware. We also briefly discuss the internal and external challenges facing static, ML-based detection methods, including our main focus in this thesis, viz. subjective and inaccurate labeling of apps.

3.4. Machine-Learning-Based Detection

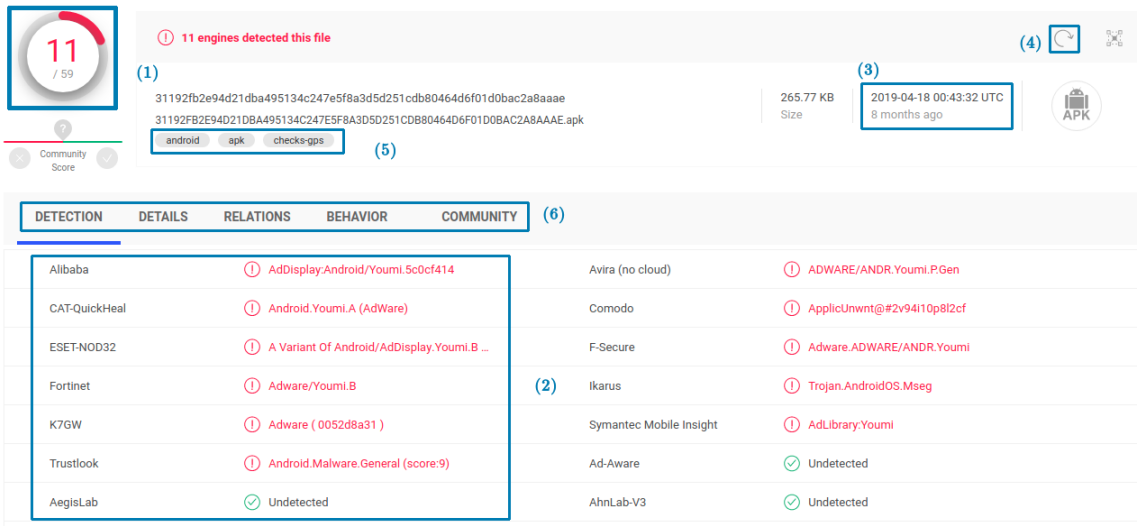
Despite being fast, lightweight, and reflecting domain knowledge, heuristic-based detection methods rely on specific assumptions and conditions to be effective. Not only does this facilitate circumventing them, but it also risks missing some characteristics of Android (malicious) apps that domain experts (e.g., malware analysts), fail to recognize. With little help from domain experts, ML-based detection methods are able to automatically find patterns and characteristics shared by Android (malicious) apps, which can facilitate correctly classifying Android apps [41, 51]. In fact, ML-based detection is the de facto detection method within the academic research community [100, 142]. Albeit using different machine learning algorithms and different types of static and dynamic features, the majority of ML-based detection methods adopt a standard process that we discuss in the following sections.

3.4.1. Data Collection and Labeling with *AndroZoo* and `VirusTotal`

The ML-based detection process starts with acquiring and labeling a collection of Android apps in the form of APK archives. In machine learning terms, this set is usually referred to as the training dataset. The typical process of acquiring Android apps is to manually crawl app marketplaces in pursuit of newly-uploaded apps, regardless of their malignancy. This process has been largely automated by platforms such as *AndroZoo* [10]. The more challenging process is that of labeling the downloaded apps as benign and malicious. Ideally, researchers should manually analyze the APK archives of the downloaded apps and monitor their runtime behaviors to discern their malignancy. Even with the subjectivity of deeming apps as malicious, as discussed in Chapter 2, manual analysis is expected to yield labels that accurately reflect the malignancy of apps. However, manual analysis cannot cope with large numbers of Android apps. So, researchers turn to platforms that provide (a) collections of Android malicious and benign apps to download, and (b) the scan results of different antiviral scanners of those apps. There is a number of online platforms that provide this service, such as *Hybrid-Analysis* [12], *Malwr* [86], and *OPSWAT* [27]. Nevertheless, the most reliable and renowned platform is *VirusTotal*. So, in this section, we focus on `VirusTotal` as the source of Android apps and their labels.

The internal structure of `VirusTotal` and how it acquires (Android) apps, how it scans them, and the agreements it has with antiviral scanners is proprietary. However, the platform provides two main methods of interaction, namely a web interface and an API.

Figure 3.1.: An illustration of the VirusTotal scan reports retrievable via the platform's web interface.



Using VirusTotal via Web Interface

VirusTotal's web interface allows users to look up the scan results of apps either by searching for them using a hash value or by uploading the executable itself (e.g., APK archive), for analysis. To avoid unnecessary analysis, the platform searches whether an app has already been analyzed, and offers the user to display the existing scan results. If the user wishes to reanalyze the app, the platform will re-scan the app using the antiviral scanners it possesses and will display, as seen in Figure 3.1, the new scan results along with different information about the queried app, which we highlight using blue squares in the figure. The platform refers to such displayed information as a *scan report*.

In the top left of the report (i.e., square **(1)**), the user can see the number of scanners deeming an app as malicious out of a total number of scanners used in the report. Details about the scanners and the labels they give to the app can be seen in square **(2)**. As mentioned in Section 1.2.1, scanners that deem apps as malicious will have their labels displayed in red next to their names, those that deem them as benign will have a green *Undetected* displayed next to their names, and scanners that did not manage to scan the app, for an unknown reason, will be grayed out. In square **(3)**, the date and time of the last time the app was analyzed is displayed. On the top right (i.e., square **(4)**), the curved arrow allows the user to reanalyze the app, which effectively re-scans the APK archive VirusTotal already possesses with the latest version of the antiviral scanners it uses.

VirusTotal allows its users to search for apps using tags, some of which are displayed in square **(5)**. Those tags are supposed to summarize the type of the app, the type of its executable, and some of its interesting functionalities. In Figure 3.1, the scan report

corresponds to an *android* APK archive that appears to use the GPS module of the device it runs on. Lastly, as highlighted by square **(6)**, `VirusTotal` provides the users with details about the internal structure of the scanned app (e.g., its components, permissions, and contacted URLs), whether it has any relations with other apps by sharing or reusing components, the API calls it issues during runtime (if available), and any available verdicts of researchers that analyzed the app.

Using `VirusTotal` via API

Despite providing the users with useful information, the web-based `VirusTotal` scan reports do not scale to provide information about Android apps in large training datasets, especially since they need to be retrieved manually. Furthermore, they are not in a machine-readable format that can help manipulate and process the information they hold (e.g., to extract numerical features). To cater to a large number of Android apps, `VirusTotal` supports retrieving the scan reports of (Android) apps using an API. By signing up to the platform, users gain access to a *public* API that limits the number of requests made to the platform's servers to a mere four requests per minute. To be able to issue more requests, users have to apply for a *private* API. Typically, access to private API keys is granted under commercial licenses. However, the platform provides access to its private API for the academic research community for a limited amount of time after submitting an application to the platform's moderators. In our case, we gained access to a limited version of `VirusTotal`'s private API for a total of one year. Under this access, we could issue up to 20,000 requests per day.

The details of how requests are handled by the platform are also proprietary. Requests can be made either using `cURL`, `Python`, or `PHP`, and are divided into three categories of requests, viz. files, URLs and domains and IPs, and comments. Using the files API requests, users can download the scan reports of apps, upload and scan their APK archives, re-scan their archives, and download new apps, as seen in Listing 3.2. Similarly, users can scan and download the scan reports of URLs and domain names. Lastly, the comments API requests enable users to download any comments made by researchers vis-à-vis an app (i.e., ones available under the Community tab in Figure 3.1).

Listing 3.2.: Examples of `VirusTotal`'s API requests to gather information about Android apps.

```
1 import requests
2
3 # Retrieve file scan reports
4 params = {'apikey': '<apikey>', 'resource': '<filehash>'}
5 response = requests.get('https://www.virustotal.com/vtapi/v2/file/report',
  ↪  params=params)
6
```

```
# Upload and scan a file
params = {'apikey': '<apikey>'}
files = {'file': ('app1.apk', open('app1.apk', 'rb'))}
response = requests.post('https://www.virustotal.com/vtapi/v2/file/scan',
 ↪  files=files, params=params)

# Re-scan a file
params = {'apikey': '<apikey>', 'resource': '<filehash>'}
response = requests.post('https://www.virustotal.com/vtapi/v2/file/rescan',
 ↪  params=params)

# Download a file
params = {'apikey': '<apikey>', 'hash': '<filehash>'}
response = requests.get('https://www.virustotal.com/vtapi/v2/file/download',
 ↪  params=params)
downloaded_file = response.content
```

Since our concern is the collection and labeling of Android APK archives, we focus on the files API requests. To acquire new APK archives, researchers can either download them from `VirusTotal` as seen in the previous listing, or from other platforms such as *AndroZoo*. To label apps, researchers usually download their `VirusTotal` scan reports, and employ different strategies to interpret the information in these reports to deem apps as malicious and benign. The information in the download scan reports resembles the information found in their web-based counterparts. However, in the API-based scan reports, the information is organized in a machine-readable format, namely JavaScript Object Notation (JSON). So, the information in the web-based scan reports can be found in the API-based ones as keys and values. For example, the number of scanners deeming an app as malicious is available under the key *positives*, the last time and date on which the app has been scanned can be found under *scan_date*, and the tags given by `VirusTotal` to the app are listed under *tags*. We summarize the `VirusTotal` keys that we use in this thesis in Table 3.2.

Labeling Apps using `VirusTotal`'s Scan Reports

As discussed earlier, the information returned by `VirusTotal`, as seen in Figure 3.1 does not directly translate into a label (e.g., malicious and benign). Users of the platform have to employ some technique to infer a label from such information. The techniques used to infer labels from `VirusTotal`'s raw information is usually referred to as labeling strategies, which we define as:

Table 3.2.: A summary of the VirusTotal scan report attributes that we use in this thesis.

Name	Description	Type	Example
first_seen	The time and date on which the app was first uploaded and scanned on VirusTotal	Formatted str	"2015-10-17 06:28:03"
last_seen	The time and date on which the app was last uploaded and re-scanned on VirusTotal	-	
positives	The number of scanners deeming an app as malicious	int	15
scan_date	The time and date on which the app was last (re-)scanned without re-submission	Formatted str	"2019-09-27 02:53:36"
scans	A collection of details about the verdicts of different antiviral scanners	dict of dicts	"Avira": {}"detected": False, "result": None, "update": "20190927", "version": "8.3.3.8"}
tags	The list tags given by VirusTotal to an app	list of strs	*contains-elf*
times_submitted	The number of times an app was uploaded to VirusTotal for analysis	int	3
total	The total number of antiviral scanners in a scan report	int	59
additional_info.androguard.Permissions	The list of permissions requested by an app as per its AndroidManifest.xml file	list of dicts	android.permission.CALL_PHONE
additional_info.positives_delta	The difference in *positives* between the current *scan_date* and the one before it	int	-5

Definition

A labeling strategy is a combination of techniques that use the attributes available in a typical VirusTotal scan report to generate a label for an app that reflects its malignancy (i.e., malicious or benign), malware type, or malware family. These techniques are usually based on the domain knowledge of the researcher employing them and/or studying the behavior of VirusTotal and the scanners it uses.

In the absence of standard labeling strategies or procedures that suggest how to devise labeling strategies, malware analysis and detection researchers are left to devise the strategies they deem fit to label Android apps based on their VirusTotal scan reports. Our survey of the literature suggests the existence of two broad categories of labeling strategies: threshold-based and what we refer to as scanner-based labeling strategies.

Threshold-based labeling strategies deem apps as malicious if the number of scanners in a scan report that deem the app as malicious meets or exceeds a value (or threshold) pre-defined by the researchers. The lower bound of the threshold is zero (i.e., no scanners deeming an app malicious), and its upper bound is the total number of scanners found in the scan report, which is usually around 60 scanners. A threshold is always a positive number, especially since a negative value does not represent any number of scanners. In addition to integer values, thresholds assume float values to represent a percentage of scanners rather than a particular number [153]. The typical scan report attributes that threshold-based labeling strategies utilize are the *positives* and *total* attributes. Lastly, threshold-based labeling strategies are usually oblivious to the scanners they base their decisions on. For example, consider the scan report in Figure 3.1; a threshold-based labeling strategy that uses the threshold of ten scanners will deem the app represented by this scan report as malicious regardless of the trustworthiness or reputation of the scanners that deem the app

as such. This property makes them susceptible to false positives, especially if the threshold value is low (e.g., one) [63].

Scanner-based labeling strategies are designed to counter this limitation of their threshold-based counterparts. So, instead of solely relying on the total number of scanners deeming an app as malicious, they base their decisions on the verdicts given by a subset of scanners that are known to be more accurate and/or trustworthy. This subset of scanners can be based on domain knowledge and expertise [15], or on experiments and measurements that reveal which scanners are more correct and consistent over time [113, 63].

In some cases, the two approaches are combined to devise a hybrid method. For example, in [15], Arp et al. focused on the verdicts of ten scanners, namely `AVG`, `Avira` (formerly AntiVir), `BitDefender`, `ClamAV`, `ESET-NOD32`, `F-Secure`, `Kaspersky`, `McAfee`, `Panda`, and `Sophos`, and deemed an app as malicious if at least two out of these scanners deemed it as such. Effectively, it is a crossover between threshold-based and scanner-based labeling strategies.

3.4.2. Feature Engineering, Selection, and Extraction

Given that machine learning algorithms operate on numerical data, the APK archives have to be processed to extract numerical and categorical features from them. The extracted features usually reflect assumptions about what constitutes and reveals the malignancy of Android apps. For example, based on observations that malicious Android apps request more permissions than their benign counterparts [75], a feature depicting the total number of permissions requested by an app might be included in the features extracted from apps in the training dataset.

The process of coming up with potential features to extract from Android apps is usually referred to as *feature engineering*. Features can be statically or dynamically extracted from APK archives. Ultimately, the result of feature extraction yields a vector of numerical features ($\hat{x}_i$) for each Android (α_i). Each feature vector is usually assigned a label (y_i) that depicts its *class*. This class can depict its nature (i.e., malicious versus benign), malware family, malware type, or other categories to which the Android app may belong to. For a group of Android apps, the collection of their feature vectors and labels is usually organized into a matrix (X) and a vector ($\hat{y}$).

Feature Selection

Feature selection aims at manually or automatically selecting a subset of more relevant or informative features from feature vectors. The relevance of a feature can be decided upon manually based on prior knowledge of the problem domain [50], or automatically using statistical algorithms. Given a matrix (X) containing feature vectors of dimensionality (n), the primary objective of such statistical algorithms is to search the 2^n subsets of features that can segregate feature vectors of different classes better than other subsets. The most straightforward search strategy is to exhaustively go over all possible feature subsets in

pursuit of the best subset. However, the exhaustive search strategy is both time- and resource-consuming [94]. So, faster and smarter strategies such as greedy hill-climbing, selection using classification models, and genetic algorithms [71] have been implemented. At the end of the search for the most relevant features, feature selection algorithms return weighted feature vectors, in which each feature is weighted according to its importance and contribution to better classification. It is up to the user to decide upon a threshold that a feature's weight needs to meet in order for its corresponding feature to be considered in the set of selected features. In special cases, the weights are either 0 or 1; features that have a weight of 1 are included in the final feature vector, whereas features assigned a weight of 0 are excluded [94].

Feature Extraction

Unlike feature selection, feature extraction algorithms do not return a subset of the features originally retrieved from, for example, Android APK archives; they rather construct new features from existing ones [51]. There are different feature extraction techniques, of which Principal Component Analysis (PCA) is the most prominent. PCA attempts to find relevant and more informative features that segregate data samples of different labels. Mathematically, the segregation is defined in terms of *variance*. A feature ($x_i^j \in \hat{x}_i$) of high variance is assumed to spread data samples in a way that facilitates classifying them. PCA works on the assumption that a large feature variance corresponds to useful information, with small variance equating to information less useful [61]. In this context, PCA reduces the dimensionality of the feature vectors while retaining most of the variation in the dataset [110]. PCA does not suggest the target dimensionality or the number of new features to consider. Deciding upon such value is application specific, and is left for the user of PCA [111]. For instance, if PCA is used to visualize the data samples, the target dimensionality is chosen to be two or three [110].

3.4.3. Training and Validation

After extraction and selection of features, the numerical representations of the Android APK archives (X) and their labels ($\hat{y}$) are used to train and validate a machine learning algorithm. This algorithm depicts the core of ML-based detection methods as it is used to decide upon the label or class (e.g., malicious and benign), of Android apps based on their numerical representations. Machine learning algorithms, often referred to as *models*, are meant to describe dependencies among data and represent the causalities and correlations between inputs and outputs [41, 51]. Given a set of observed data (X) and a learning model $F(X, \theta)$, the objective of machine learning is to estimate a set of parameters θ that minimize the learning error $E(F(X, \theta), X)$. The learning error depicts the difference between the outputs produced by the learning model $F(X, \theta)$ and the ground truth ($\hat{y}$). The learning process, also known as the *training* process, is considered complete once the learning error converges to a minimum value.

The learned models can perform a variety of tasks such as classification, regression, and clustering [41, 21]. Within the domain of malware analysis and detection, both classification and clustering can be utilized. For example, in some cases, researchers aspire to group (malicious) apps together to identify families or types [153]. In typical detection scenarios, however, the main objective is to assign an app (α^*) a label that depicts, for instance, whether it is malicious or benign; this task is called classification.

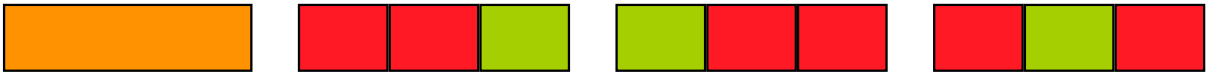
The choice of the model to train largely depends on the training dataset (X) being labeled or not. In the case of classification, the labels of apps in the training dataset are usually known via the ($\hat{y}$) vector. If so, the training process is known as *supervised learning*. Since every feature vector ($\hat{x}_i \in X$) has a corresponding label ($y_i \in \hat{y}$), the training dataset can be represented as $D = \{(\hat{x}_1, y_1), (\hat{x}_2, y_2), ..., (\hat{x}_n, y_n)\}$.

The exact technicalities of the training process differ from one model to another (e.g., decision tree versus support vector machines). However, in essence, the training process attempts to estimate a *decision boundary* that best separates the feature vectors belonging to different labels or, more technically, minimizes the learning error $E(F(X, \theta), X)$. Depending on the trained model, this decision boundary can be a line equation, a multi-dimensional plane, or in the case of models such as K-Nearest Neighbors (KNN), a majority vote function.

The trained model is used to assign labels to apps that have not been used during training (i.e., test or out-of-sample apps). The first step to label such apps is to represent them as vectors of numerical features ($\hat{x}^*$) resembling the features used to represent apps in the training dataset. Using ($\hat{x}^*$), the trained model estimates the side of the decision boundary on which the vector resides and, in turn, its estimated label. In our scenario, this side can be the side of malicious apps versus that of benign apps or the side of the malware family `Airpush` versus that of `SimpleLocker`.

The process of estimating the labels of apps never used in training is referred to as *testing*, and should mimic the conditions that the trained models go through after being deployed. The more accurate the training model can label out-of-sample apps, the better it is said to *generalize*. If it cannot generalize well, the trained model is usually said to *overfit* to its training dataset. That is, the model learns characteristics specific to the feature vectors in its training dataset that are not necessarily to be found in out-of-sample feature vectors. To give an example within the domain of Android malware detection, consider a scenario in which a ML model is trained using one feature depicting the size of each APK archive in the training dataset. The trained ML model will learn to separate malicious and benign Android apps based on the sizes of their APK archives. If the majority of apps were developed during the same time period (e.g., 2010), the trained ML model would learn the sizes during that time period. While the trained model is expected to effectively classify Android apps developed in 2010, it is expected to fail to segregate newer apps based on the sizes of their APK archives, given that Android apps significantly grew in size between 2010 and 2019. In this case, the trained model is said to have overfitted to its training dataset.

To ensure that the trained models generalize well before deployment, researchers simulate the testing process by training a model using a subset of feature vectors in the training dataset and testing the trained model using the remaining subset of feature vectors. This



(a) Original Dataset (b) First Iteration (c) Second Iteration (d) Third Iteration

Figure 3.2.: This figure depicts 3-Fold cross-validation. Figure (a) shows the original dataset that is divided into three equal folds. In figure (b), the first two folds, in red, are used for training and the remaining fold, in green, is used in testing. In the second iteration, figure (c), two different folds are used for training. Figure (d) concludes the validation process as all fold combinations have been exhausted.

simulation process is referred to as *validation*. The conventional method of performing validation is to split the training dataset into two-thirds for training, and one-third for validation [143]. However, this method depicts only one scenario of what the trained model can encounter during training and testing, which can give a false estimation of the generalization capabilities of the trained model. To make the validation process more objective and comprehensive other methods are used to perform validation, including K-Fold Cross-Validation.

K-Fold Cross-Validation K-Fold Cross-Validation divides the training dataset $D = \{(\hat{x}_1, y_1), (\hat{x}_2, y_2)..., (\hat{x}_n, y_n)\}$ into (K) equal portions, or *folds*. Given a machine learning model, K-Fold Cross-Validation iterates over (D), systematically or randomly picking $(K-1)$ folds as the training dataset and the remaining fold as the test dataset of the model. At every iteration, the classification accuracy of the model is calculated. This process is carried out for (K) iterations. At the end of the iterations, the recorded (K) accuracies are averaged to yield the overall accuracy of the classifier on the dataset (D). Figure 3.2 gives an example of 3-Fold cross-validation. K-Fold Cross-Validation is an intuitive, efficient, and objective methodology to assess the generalization ability of a model.

3.4.4. Decision Boundaries of Learning Algorithms

In the previous section, we mentioned how different ML classifiers learn how to separate feature vectors that belong to different classes and uses this separation to classify out-of-sample feature vectors. Furthermore, we discussed that this separation, which we referred to as a decision boundary, can assume different formats such as an equation of a line or a multi-dimensional plane. The question we address in this section is whether the difference in the format of the decision boundaries learned by different ML classifiers may have an impact on their classification abilities.

To facilitate understanding the concepts we introduce in this section, we use the visualization in Figure 3.3. In this figure, we plot the decision boundaries learned by the ML classifiers we use later in this thesis to separate two classes in three synthetic datasets (i.e., red circles and green triangles). The three datasets resemble different scenarios of how the feature

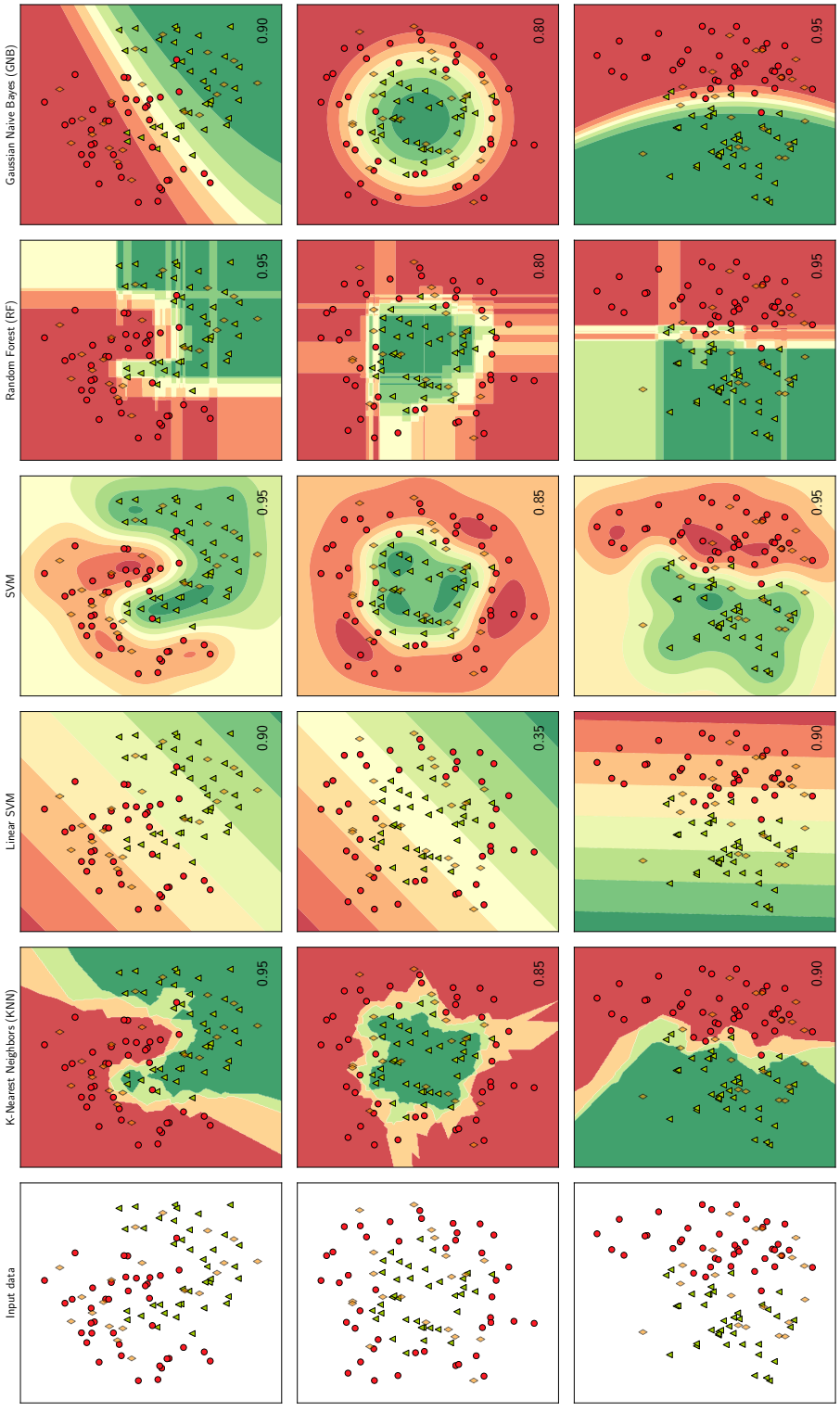


Figure 3.3.: An illustration of how different ML classification algorithms separate feature vectors and the impact such separation has on their ability to correctly classify out-of-sample feature vectors [125]. In this illustration, the red circles and green triangles depict training feature vectors belonging to two different classes, whereas the orange diamonds depict out-of-sample test feature vectors. The numbers on the bottom right of each figure depict the percentage of out-of-sample feature vectors the classifier managed to correctly classify (i.e., accuracy).

vectors belonging to different classes occupy the same dimensional space in relation to one another. That is, in the bottom dataset, feature vectors that belong to either class are clustered together in two large groups, which might facilitate separating them via a straight line drawn between them. Similarly, the feature vectors in the top dataset can be separated via a higher-dimensional line, especially since they appear to be more intertwined. However, in the middle dataset, the feature vectors of one class (i.e., green triangles), are surrounded by the vectors belonging to the other class, which makes it difficult for some ML classifiers to separate. In this context, the feature vectors in the top and bottom datasets are said to be *linearly separable*. The goal of the classifiers in Figure 3.3 is to classify the test feature vectors depicted as orange diamonds.

There are two factors that impact the shape of the decision boundary learned by ML classifiers to separate the training feature vectors. Firstly, ML classifiers are designed to separate feature vectors in different manners. For example, the linear SVM classifiers, which are used by *Drebin*, attempt to draw lines between the two classes. Random forests use their tests and splits to compartmentalize the training dataset. That is, after each split, the decision trees in the forests isolate the segment of the training dataset that it managed to capture its ground truth, and perform further tests and splits on the remaining segment. The second factor that affects the shape of the decision boundaries is the organization of the feature vectors in the dimensional space. As part of separating different classes, ML classifiers attempt to identify regions of the dimensional space occupied by feature vectors of a specific class. In those regions, the classifier is confident that a test feature vector is likely to belong to the same class. This confidence is depicted by the different degrees of red, green, and yellow colors in Figure 3.3. As seen in the figure, regions in which green triangles are the majority have darker green colors, while regions in which the red circles are the majority have darker red colors. The more mixed the feature vectors are, the lighter the color of that region.

It follows that the ability to correctly classify out-of-sample feature vectors will differ from one ML classifier to another and from one dataset to another. In this example, we use the accuracy metric as our measure of each classifier's performance. Shown in the bottom right corner of each subfigure, accuracy measures the percentage of out-of-sample apps that each classifier managed to classify correctly. Taking the linear SVM classifier as an example, one can notice that the classifier can accurately classify out-of-sample apps in the datasets where the training feature vectors are linearly separable (i.e., top and bottom datasets). However, the classifier's performance drops to 0.35 for the middle dataset, in which the apps are not linearly separable. Similarly, the accuracy of the Gaussian Naive Bayes (GNB) classifier is better on the bottom dataset than on the top dataset, whereas the accuracy of Random Forest (RF)s is better on the top dataset than on the bottom one.

We relate this example to the problem of Android malware detection using ML-based detection methods and this thesis as follows. The performance of ML-based detection methods hinges on how well the ML classifier can separate malicious and benign apps. What we demonstrated in this example is that some ML classifiers can cope with linearly-inseparable data, while others cannot, which explains the discrepancies in their performance

across different datasets. This hinges on the type of features extracted from the APK archives, which dictate how the feature vectors will occupy the dimensional space in relation to one another. If the separability of the aforementioned feature vectors coincides with the chosen ML classifier, the performance of the ML-based detection method is likely to be high. In other words, there are no ML classifiers that will always perform well on any types of features extracted from Android apps. This partially explains the results in Section 6.4: depending on the type of features extracted from the APK archives of apps in the *AndroZoo* dataset, some ML classifiers can perform better than others. Furthermore, the labels assigned to apps in this dataset–which hinges on the employed labeling strategy–impact the distribution of the feature vectors representing malicious and benign apps in the dimensional space. This impacts the decision boundaries learned by different ML classifiers, effectively altering their detection performance.

3.5. Challenges Facing ML-Based Detection

The authors of ML-based Android detection methods usually report promising detection results. However, as discussed in Chapter 1, Android malicious apps continue to evade detection and can be found on app marketplaces. In addition to the theoretical and practical limitations of malware detection that we discussed earlier in this chapter, there are three main challenges that face ML-based malware detection in general and Android malware detection in particular. These challenges are related to the method Android apps are labeled prior to training a detection method, the deterioration of the ML-based detection methods' performance over time, and proactive malware authors that possess information about the internal structure of ML-based detection methods and design their malicious apps to evade detection by such methods (i.e., adversarial ML).

3.5.1. The Choice of Features and Classifiers

In Section 3.4.4, we discussed using an example that the performance of ML classifiers and, in turn, that of ML-based detection methods depends on the ability of the chosen ML classifier to separate the feature vectors of malicious and benign apps. It follows that the type of features extracted from the apps' APK archives plays a vital role in enhancing the performance of such detection methods. So, the main challenge that faces malware analysis and detection researchers is to identify or devise numerical features to be statically or dynamically extracted from Android apps, and match those features to a ML classifier that can separate them well and, thus, effectively classify out-of-sample Android apps. Surveying the literature, we found that there is no consensus on which type of features to use with which classifiers [142, 16]. Instead, the majority of ML-based detection methods introduce new types of features that are either matched with specific classifiers, or are evaluated using different classifiers [87, 104, 156, 97, 43, 15].

3.5.2. The Subjectivity of Malware Labeling

The accuracy of the labels that correspond to feature vectors used to train a machine learning model plays a vital role in the model's ability to separate feature vectors in the training dataset and, subsequently, generalize to out-of-sample feature vectors. Recall that machine learning models attempt to find a decision boundary that best separates two, or more, classes of feature vectors. If the labels of such feature vectors change, the drawn decision boundary will also change to cater to the new labels. Consider the illustration in Figure 3.4, which shows a SVM trying to separate two classes of feature vectors in a 2-dimensional space. In this illustration, we consider the circles to be feature vectors of malicious apps, the triangles to be the feature vectors of benign apps, and the diamond to be the feature vector of an out-of-sample malicious app that we use to validate the quality of the trained SVM. Those shapes change color according to the strategy adopted to label them prior to training the classifier. For example, the difference between the left and middle subfigures in Figure 3.4 is that the label of the right outermost triangle changed from benign (i.e., green) to malicious (i.e., red). Changing the label of this app and its feature vector forces the SVM to estimate a new decision boundary to separate the classes according to the new labels. Similarly, the subfigure on the right has a different decision boundary to cater to the different labels.

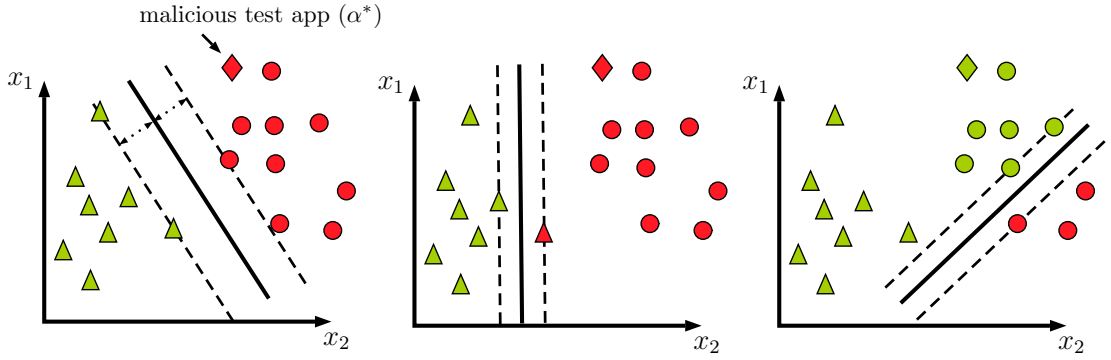


Figure 3.4.: An illustration of how the decision boundary learned by a SVM will differ in an attempt to cope with different labels.

The main problem in this example is not in estimating a different hyperplane or decision boundary; instead, it is a problem of training a SVM that accurately represents the ground truths of the training feature vectors and can generalize well. In the left subfigure, the SVM can decently separate malicious and benign apps, which helps train a model that correctly recognizes the malignancy of the test app (α^*) depicted as a diamond. The SVM trained in the middle subfigure also recognizes the malignancy of the test app; nevertheless, it trains a model with narrow margins between the positive and negative hyperplanes, which might misclassify benign test apps as malicious, resulting in a potentially high rate of false positives. The decision boundaries in the right subfigure depict the opposite case, in which

malicious apps are mistakenly labeled as benign, resulting in a SVM that misclassifies (α^*). With this hypothetical example, we wish to convey the importance of labeling apps and their feature vectors as accurately as possible.

Assuming an objective and unbiased expert perspective, accurate labeling of Android apps can be achieved by manually analyzing and reverse engineering apps, as seen in Section A. However, a fully manual method cannot cope with the number of apps uploaded daily to marketplaces, such as Google Play. Private firms (e.g., antiviral software firms) are said to rely on semi-automatic methods to analyze apps, especially since they have the resources to recruit malware analysts and researchers that develop such methods. Academic researchers, however, do not possess such resources and rely on the results returned by antiviral software. To be objective, researchers attempt to rely on the scan results of as many antiviral scanners as possible. As discussed earlier, one platform that is renowned within the academic community that provides such a service is `VirusTotal` [148]. This online platform displays to users the scan results of around 60 antiviral scanners (hereafter scanners), without any bias towards a certain subset of scanners or recommendations on which scanners to focus on. In this context, it is completely up to the user to decide upon the scanners they consider in labeling an Android app.

The most common strategy adopted to label apps according to their `VirusTotal` scan reports is one that requires a minimum number of scanners to deem an app malicious. If the number of scanners on `VirusTotal` that deem an Android app as malicious surpasses such a minimum number, the app is labeled as malicious; otherwise, the app is deemed as benign. Effectively, this minimum number is a *threshold*. So, throughout this thesis, we refer to this type of labeling strategy as *threshold-based* labeling strategies. Unfortunately, there are no standard methods of determining the threshold of `VirusTotal` scanners that help to accurately label Android apps, which encourages researchers to adopt different labeling strategies that conform with their understanding or definitions of malignancy. As seen in Figure 3.4, different labeling might impact the quality and effectiveness of a ML-based detection method, especially since it significantly alters the composition of the training and test datasets.

To demonstrate the impact of varying the threshold of `VirusTotal` scanners on the composition of a dataset, we used different thresholds adopted by the academic community to label Android apps in different datasets according to their `VirusTotal` scan reports. In particular, we used thresholds of one [75], four [100], ten [150], and 50% [153] `VirusTotal` scanners to deem apps as malicious. As mentioned earlier, the term *positives* refers to the number of `VirusTotal` scanners deeming an app as malicious. As seen in Table 3.3, the impact of varying the threshold of positives on the composition of a dataset varies from one dataset to another. For example, the *AMD* dataset maintains the same composition among the thresholds of one, four, and ten before radically changing under a threshold of 50% of scanners. Ironically, a threshold of 50% of scanners is the threshold the authors of the *AMD* dataset used to deem apps as malicious.

One observation we can make by examining the data in Table 3.3 is that a threshold of one `VirusTotal` scanner seems to be biased towards labeling Android apps as malicious,

Table 3.3.: The impact of varying the threshold of VirusTotal scanners (i.e., *positives*), used to deem Android apps as malicious on the composition of different datasets we use in this thesis. Apps were labeled using VirusTotal scan reports downloaded on November 8^{th}, 2019.

Datasets \ Thresholds	*positives* ≥ 1 [75]		*positives* ≥ 4 [100]		*positives* ≥ 10 [150]		*positives* ≥ 50% [153]		Total Apps
	malicious	benign	malicious	benign	malicious	benign	malicious	benign	
AMD	24,553 (100%)	0 (0%)	24,553 (100%)	0 (0%)	24,553 (100%)	0 (0%)	5,492 (22.36%)	19,061 (77.63%)	24,553
Piggybacking	1,820 (66.1%)	925 (33.6%)	1,116 (40.5%)	1,638 (59.47%)	998 (36.23%)	1,756 (63.76%)	58 (2.1%)	2,696 (97.9%)	2,754
GPlay	127 (<1%)	24,035 (99.48%)	5 (<1%)	24,157 (99.97%)	1 (<1%)	24,161 (≈100%)	0 (0%)	24,162 (100%)	24,162
AndroZoo	2,130 (34.50%)	4,043 (65.50%)	1,178 (19.08%)	4,995 (80.91%)	159 (2.57%)	6,014 (97.42%)	5 (<1%)	6,167 (≈100%)	6,172
Hand-Labeled	33 (33%)	67 (67%)	23 (23%)	77 (77%)	21 (21%)	79 (79%)	2 (2%)	98 (98%)	100
Hand-Labeled 2019	13 (13%)	87 (87%)	6 (6%)	94 (94%)	4 (4%)	96 (96%)	0 (0%)	100 (100%)	100

whereas a threshold of 50% of scanners seems biased towards labeling them as benign. However, only considering the thresholds of four and ten scanners does not seem to stabilize the composition of datasets such as *Piggybacking* and *AndroZoo*. Seeing how the composition of a dataset changes across different thresholds and given our discussion of how sensitive ML-based detection methods are to labels, it is expected that different researchers adopting different thresholds will get different performances from their detection methods even if they are using the same dataset and the same machine learning algorithm.

Different thresholds seem to depict different views of malignancy and risk levels adopted by researchers. For example, a threshold of only one VirusTotal scanner might be adopted by a very cautious researcher who considers Android apps as benign if and only if no VirusTotal scanners deem them as malicious. A more lenient researcher might opt to use a threshold of 50% of VirusTotal scanners to deem apps as malicious in order to, for example, avoid ambiguous malware types such as Adware and Riskware. Nevertheless, without having a standard or systematic method of labeling Android apps according to their VirusTotal scan reports, it is difficult to assess the effectiveness of different ML-based detection methods released by the research community.

3.5.3. Performance Decay over Time

Similar to benign apps, Android malicious apps adopt the same technologies, resources, and trends to generate monetary profit or deliver the malicious intents of their authors. For example, in the late 1990s and early 2000s, a large number of services relied on premium SMS numbers. Malware authors saw this as an opportunity to make a profit, and entire malware families were implemented to deplete users' mobile phone credits by secretly sending SMS messages to premium numbers owned by the malware authors [107, 153, 108]. Nowadays, malware authors target newer technologies such as cryptocurrencies and digital marketing, which are respectively targeted by the malware families XLoader [52] and

`SimBad` [26]. Other recent malware families, such as `RedDrop`, make use of the high-speed internet connections Android devices usually have access to, and upload recorded audio to the cloud-storage accounts on Google Drive and Dropbox [6].

Depending on the type of features extracted from the APK archives, the feature vectors representing apps belonging to those new malware families may look different from those belonging to older families (e.g., because the type of API calls used by malicious apps changed over time). From the perspective of an ML-based detection method, those new feature vectors cannot be matched to many or enough feature vectors of malicious apps used to train the method, potentially forcing the detection method to label those apps as benign. Ultimately, this *concept drift* in the functionality and appearance of malicious apps [62] is expected to impact the detection performance of once effective detection methods negatively.

In [44, 100], Pendlebury et al. refer to this problem as *temporal experimental bias*. They demonstrated that even the most renowned of detection methods, such as *Drebin* [15] and *MaMaDroid* [87], suffer from performance decay over time. This problem of performance decay does not necessarily relate to the mediocrity of the trained ML-based detection methods. It is a natural phenomenon that occurs as Android apps evolve, including malicious ones. In other words, there are no clear issues in the ML-based detection methods that need to be addressed by researchers. Pendlebury et al. suggested the frequent re-training of ML-based detection methods to cope with the release of not only new malicious apps but with newer versions of old malware families (e.g., obfuscated versions).

3.5.4. Adversarial Machine Learning

Adversarial machine learning is a relatively new challenge that faces machine learning algorithms in general, including ML-based detection methods [44, 49, 156]. In essence, adversarial machine learning focuses on altering training or test data points (e.g., APK archives), to force machine learning algorithms to make false predictions [44]. Formally, this requires crafting a fake feature vector of ($\hat{x}$), designated ($\hat{x}^f$), that confuses the trained model (F), which can be formally presented as the following problem [141]:

$$\hat{x}^f = \hat{x} + \delta_x = \hat{x} + min||z|| s.t.\ F(\hat{x} + z) \neq F(\hat{x})$$

This equation means that adversaries aspire to add some minimal perturbation (δ_x) or (z) to the original feature vector ($\hat{x}$) that makes the model (F) misclassify the feature vector. The nature of this perturbation differs from one ML algorithm to another [156], which requires knowledge of the internal mechanics of the targeted machine learning algorithm [49]. Given an ML-based Android malware detection method (F) that takes in a feature vector ($\hat{x}$) and returns $y' = 0$ indicating that ($\hat{x}$) is benign and $y' = 1$ indicating that ($\hat{x}$) is malicious, assume that the adversary (i.e., malware author), can use (F) as an *oracle*. That is, the adversary can send (F) an arbitrary feature vector ($\hat{x}^f$) and observe its output $y' = F(\hat{x}^f)$

an unlimited number of times. With the knowledge of the feature set used by (F), the adversary is expected to modify the values in ($\hat{x}^f$) and send it to (F) until the result $y' = 0$.

In practice, Android malware authors usually do not have access to information such as which machine learning algorithm is being used by the ML-based detection method, or which feature set is being used by the algorithm. So, the closest that such malware authors can get to adversarial machine learning is to guess which features are being used by the detection algorithm and which features are more influential than others. Using such information, malware authors can trace such features to the functionalities of their apps, and modify or obfuscate them in a manner that will alter their values in the feature vectors and confuse the machine learning model. Another possible technique is to repackage the benign apps a ML-based detection method is presumed to utilize as examples of benign apps, so that the resulting feature vectors of the repackaged, malicious apps look similar to their original, benign counterparts and, hence, be classified as benign.

3.6. Summary

Despite the theoretical and practical limitations that face it, the problem of automated (Android) malware detection has been researched for decades. In essence, the problem of automated malware detection is that of matching an app to a group of apps that depict its class, family, or type. Focusing on Android malware, we categorized different approaches to Android malware detection in terms of the representation of Android apps (i.e., static versus dynamic), and the techniques used to match apps to other apps (i.e., heuristic-based and ML-based). Given its popularity, we focused on ML-based detection methods and discussed how machine learning could be used to analyze and detect Android malware. Furthermore, we discussed the main challenges facing this type of Android malware detection. Each of the four challenges we discussed has its own line of research, which makes it difficult to focus on all of them. In this thesis, we focus on the problem of labeling Android apps. In the following chapters, we introduce methods to reliably and accurately label Android apps according to their `VirusTotal` scan reports and study the impact of such methods on the performance of ML-based detection methods.

Part II.

Accurate Labeling for Effective Detection

4. Threshold-Based Labeling Strategies

This chapter focuses on threshold-based labeling strategies, their structure, advantages, and disadvantages. It reveals their sensitivity to the dynamicity of `VirusTotal`*, unveils some limitations of the platform, and how to work around them to choose more stable thresholds. Parts of this chapter have previously appeared in peer-reviewed publications [9], and [116], co-authored by the author of this thesis.*

Threshold-based labeling strategies are the de facto method to interpret `VirusTotal` scan reports to label apps. However, there are not standard procedures on how to design these methods, which forces researchers to use their intuitions and choose thresholds based on their domain knowledge and how they perceive malignancy. Given their popularity, in this chapter, we delve into studying threshold-based labeling strategies. In (Section 4.1), we discuss how different thresholds might be chosen. Section 4.2 compares the labeling accuracy of the threshold-based labeling strategies that are widely used within the literature. In Section 4.3, we discuss how the dynamicity of `VirusTotal` impacts the performance of these threshold-based labeling strategies and unveils the first limitation of `VirusTotal`. Lastly, in Section 4.4, we provide actionable insights about the usage of threshold-based labeling strategies by discussing how to find the currently optimal threshold to use in labeling Android apps.

4.1. Choosing a Threshold

As mentioned in Section 3.4.1, users of threshold-based labeling strategies adopt a number between one and the total number of scanners typically found in a `VirusTotal` scan report (i.e., around 60), as their threshold of *positives*. Despite the freedom of choosing the threshold's value, in the literature, one can notice the adoption of only a small subset. In this section, we attempt to identify the reasons behind choosing some specific values as the threshold for threshold-based labeling strategies.

Surveying the literature, we found two approaches to choosing a value for a threshold. In the majority of cases, a threshold is chosen as a combination of (a) the researchers' approach to deeming apps as malicious being more cautious or lenient, and (b) what appears to be familiar human concepts. For example, a threshold value of one [113, 75] reflects a more cautious approach to deeming apps as malicious. That is, once an app is deemed as malicious by **any** `VirusTotal` scanner, researchers that adopt this strategy will deem

the app as malicious to avoid what they may believe is a risk of a false negative by other scanners. Researchers that are more concerned with false positives or ones that wish to filter out ambiguous malware types, such as `Adware`, may adopt higher thresholds, such as 50% of scanners [153]. A threshold of 50% of scanners translates to around 30 `VirusTotal` scanners, which is a relatively large number of scanners that agree upon the malignancy of an app compared to a threshold of one scanner. Given the discussion in Section 2.5 about the subjectivity of malware labeling and that around 51.5% of `VirusTotal` scanners cannot recognize the malignancy of apps in the *AMD* dataset, a threshold of 50% seems to be an exaggerated one. In this case, we argue that the chosen threshold is a combination of a more lenient approach to label apps as malicious and the concept of a simple majority (i.e., 50% of votes + 1). Within the literature, there are other thresholds, such as five [7] and ten [150, 58], that were adopted without adequate justification. Is it because the numbers five and ten are relatable (e.g., five fingers in a human hand)?

The second approach to choosing a threshold is based on studying `VirusTotal` and its scanners over a period of time. Unfortunately, this approach represents a minority of cases. For example, in [91], Miller et al. concluded their experiments that a value of four depicts the optimal threshold to label Android apps.

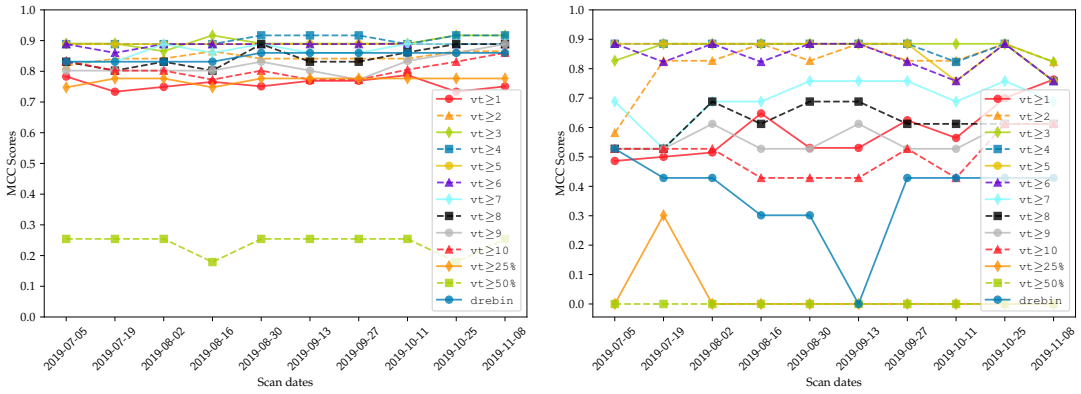
4.2. Labeling Accuracy of Threshold-based Labeling Strategies

The main objective of labeling strategies is to assign labels to apps based on their `VirusTotal` scan reports that accurately represent their ground truths. In this section, we discuss the ability of threshold-based labeling strategies to accurately label apps based on their `VirusTotal` scan reports and identify the aspects of `VirusTotal`'s dynamicity, if any, and their impacts on the accuracy of those labeling strategies. To do that, we need datasets that are (a) diverse (e.g., in terms of age and sources), and (b) have an accurate ground truth. We use the two test datasets that were discussed in Section 1.3.4, namely *Hand-Labeled* and *Hand-Labeled 2019* because they were randomly crawled by *AndroZoo* from different app marketplaces, were developed between 2010 and 2019, and because we manually labeled them, which we assume makes their ground truth reliable.

We focus on threshold-based labeling strategies that have been utilized by researchers to train and evaluate novel malware detection methods. In particular, we use thresholds between one and ten scanners [75, 113, 44, 91, 7, 150, 58], 25% of scanners, 50% of scanners [153], and the strategy adopted by the authors of *Drebin*[1] [15]. For readability, labeling strategies that use a numerical value for a threshold, say (τ), are referred to as $\texttt{vt} \geq \tau$. For example, the labeling strategy that uses a threshold of one scanner is referred to as $\texttt{vt} \geq 1$.

In Figure 4.1, we plot the performance of each labeling strategy on the *Hand-Labeled* and *Hand-Labeled 2019* datasets between July 5^{th}, 2019 and November 8^{th}, 2019 in terms of

[1] An app is deemed as malicious if at least two out of the following ten `VirusTotal` scanners label it as such: `AVG`, `Avira` (formerly AntiVir), `BitDefender`, `ClamAV`, `ESET-NOD32`, `F-Secure`, `Kaspersky`, `McAfee`, `Panda`, and `Sophos`.



(a) *Hand-Labeled* (b) *Hand-Labeled 2019*

Figure 4.1.: The labeling accuracy of different threshold-based labeling strategies against apps in *Hand-Labeled* and *Hand-Labeled 2019* datasets based on their `VirusTotal` scan reports downloaded between July 5^{th}, 2019 and November 8^{th}, 2019. The labeling accuracy is calculated in terms of the MCC score of each labeling strategy.

the Matthews Correlation Coefficient (MCC) score [129]. For example, in Figure 4.1a, the labeling strategy `vt`$\geq$`1` had an MCC score of about 0.79 based on scan reports downloaded on July 5^{th}, 2019. Conventional metrics, such as accuracy, are unable to capture or penalize bias towards certain classes in imbalanced datasets. For example, using the accuracy metric ($\frac{TP+TN}{P+N}$), the `vt`$\geq$`50%` would have scores of 0.78 and 0.90 for the *Hand-Labeled* and *Hand-Labeled 2019* datasets, effectively rewarding its bias towards the benign class that comprises the majority of apps in those datasets. However, MCC is capable of capturing such a bias of this labeling strategy towards classifying apps as benign and gives a value that objectively describes the strategy's performance. The MCC values range from -1 (i.e., all apps were misclassified) to 1 (i.e., perfect classification), with the value of 0 indicating a classification ability similar to random classification.

Even within this small period of four months, one can notice that some threshold-based labeling strategies are more accurate than others. Starting with the performance of `vt`$\geq$`1`, while the labeling strategy managed to achieve a decent MCC score on apps in the *Hand-Labeled* dataset, its performance noticeably decreased against apps in the newer *Hand-Labeled 2019* dataset. As discussed earlier, low threshold values, such as one or two scanners, might result in false positives, especially against new apps whose `VirusTotal` scan reports are not mature enough. In rare cases, only a few scanners can recognize the malignancy of apps. However, in most cases, if an app has one or two `VirusTotal` scanners deeming it as malicious, then it is most likely to be a false positive. For example, we noticed that some scanners such as `Tencent` consistently label any apps

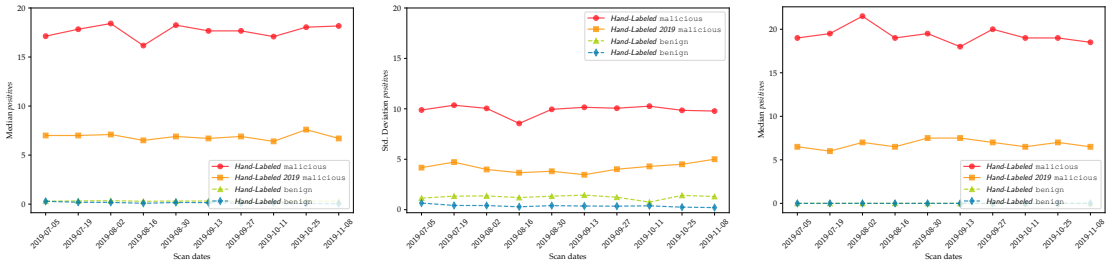
(e.g., ed23237e34ff47580a99ac70f35e84b32c05ab1d), that utilize *App Inventor*[2] as malicious apps belonging to the A.gray.inventor.a malware family.

As for (vt≥50%), pushing the threshold that high might prevent recently-developed malicious apps and apps that belong to ambiguous malware types (e.g., Adware) from being labeled as malicious, resulting in a high number of false negatives. Similar to vt≥1, the older the app and its VirusTotal scan report, the better the performance of vt≥50% and the newer the app, the worse the performance, especially since the malicious apps were not labeled as malicious by enough VirusTotal scanners to make the 50% mark required by the strategy to successfully deem them as malicious. In Section 2.5, we found that apps in the *AMD* dataset had an average detection rate of 51.5%, and hypothesized that this might negatively impact the performance of threshold-based labeling strategies that use a threshold of 50% of VirusTotal scanners to deem apps as malicious. The performance of vt≥50% indeed supports this hypothesis.

Another aspect of how the age of apps and, in turn, the maturity of their VirusTotal scan reports impacts the performance of different threshold-based labeling strategies can be seen in the proximity of different MCC lines in Figure 4.1. In particular, in Figure 4.1a, the lines of almost all threshold-based labeling strategies are close to one another and exhibit a relatively steady performance (i.e., the performance does not noticeably fluctuate). However, the MCC lines in Figure 4.1b are more distributed across the figure and exhibit more fluctuations in performance. For example, on the *Hand-Labeled 2019* dataset, the MCC score of drebin sharply decreased from a little above 0.3 on August 30th, 2019 to almost 0.0 on September 13th, 2019 only to sharply increase to around 0.45 two weeks later. The reason behind the proximity in the case of apps in the *Hand-Labeled* dataset is that their *positives* values are high enough to accommodate threshold values up to at least 15 scanners, which is represented by the vt≥25% labeling strategy. The novelty of malicious apps in the *Hand-Labeled 2019* means that their *positives* values are much lower in comparison, which prevents thresholds higher than six scanners from achieving high MCC scores. This claim can be verified upon examining the mean, median, and standard deviation of the *positives* attribute for malicious and benign apps in both datasets over the same period of time. As seen in Figure 4.2, the *positives* attribute of malicious apps in the *Hand-Labeled* dataset stays within the range of 15 to 20. Even with a standard deviation of ten scanners, the range of scanners needed to label malicious apps in this dataset correctly remains between 7.6 and 20 scanners. As for the malicious apps in the *Hand-Labeled 2019* dataset, their *positives* values have mean and median values around seven, ranging between 2.89 scanners and 10.91 scanners. The benign apps in both datasets have mean and median values that are almost zero with a negligible standard deviation of at most one scanner. So, any threshold values above three are guaranteed to avoid false positives.

In Figure 4.1, we notice that only a subset of values of the *positives* attribute allows the labeling strategies using thresholds between three and six VirusTotal scanners (i.e.,

[2]App Inventor is a visual programming environment maintained by MIT that enables non-technical users to develop apps for Android [60].



(a) Mean *positives* (b) Std. Deviation *positives* (c) Median *positives*

Figure 4.2.: The mean, standard deviation, and median of the *positives* attributes found in scan reports of apps in the *Hand-Labeled* and *Hand-Labeled 2019* datasets between July 5^{th}, 2019 and November 8^{th}, 2019.

vt$\geq$3, vt$\geq$4, vt$\geq$5, and vt$\geq$6), to outperform all other threshold-based labeling strategies on both datasets in terms of the MCC score. Recall that the *positives* attribute is the number of VirusTotal scanners that deem an app as malicious. So, the question is, why does a range of VirusTotal scanners between three and six achieves the best MCC scores on both datasets.

We hypothesize that there might be a number of VirusTotal scanners *at least* between three and six that synchronize the labels they assign to apps within the same period of time either because they the same detection engine [157, 22] or because they copy one another's verdicts [88]. We used association rule mining to investigate whether there is a group of VirusTotal scanners that collectively and correctly label malicious apps. Association rule mining attempts to associate items that usually occur together according to a history of transactions [81]. For instance, given three separate transactions of items bought in a supermarket (Milk, Eggs, Yogurt), (Milk, Apple, Corn), and (Milk, Eggs, Beer), association rule mining can conclude that the items Milk and Eggs were bought together 67% of the time, which is referred to as *support*. We retrieved the list of VirusTotal scanners that correctly deem apps in the *Hand-Labeled* and *Hand-Labeled 2019* datasets as malicious according to our manually-assigned ground truths. We found **no** evidence that the **same** set of three to six VirusTotal scanners **always** correctly deem apps as malicious in either dataset. However, some VirusTotal scanners correctly deemed apps in both datasets as malicious more often than others. For example, for 95% of the malicious apps in the *Hand-Labeled* dataset, the scanner ESET-NOD32 deemed them as malicious; ESET-NOD32 also deemed 89% of the malicious apps in the *Hand-Labeled 2019* dataset as malicious. Similarly, the scanner SymantecMobileInsight correctly deemed 82% and 89% of malicious apps in the *Hand-Labeled* and *Hand-Labeled 2019* datasets as malicious. Other VirusTotal scanners occasionally appeared alongside those two scanners to correctly deem apps in both datasets as malicious. This results into mining association rules such as {ESET-NOD32, SymantecMobileInsight, Cyren}, {ESET-NOD32, SymantecMobi

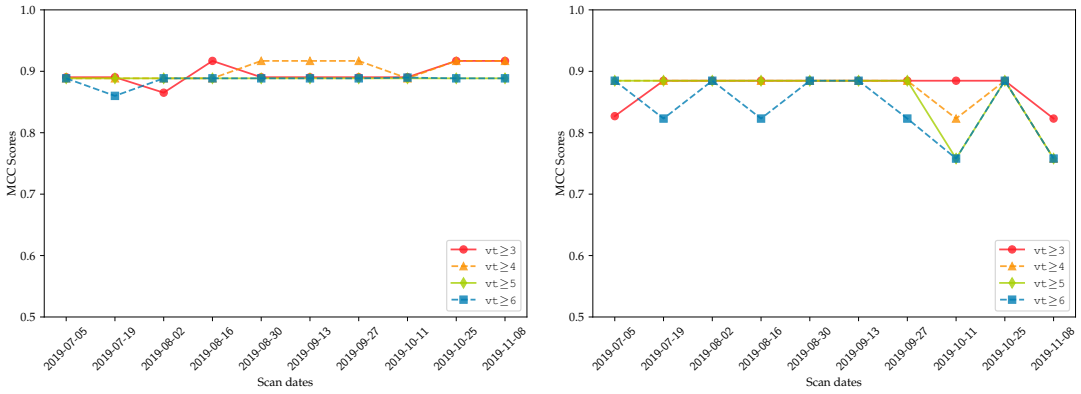
leInsight, Fortinet}, and {ESET-NOD32, SymantectMobileInsight, Ikarus}, etc. In summary, the rules mined from both datasets suggest that for each app, different VirusTotal scanners appear alongside more reliable, consistent scanners to deem an app malicious. Despite this change in scanners, for 67% of the apps in both datasets, the mined rules contained an overall number of scanners that fell within a range of three and six. The remaining 33% of apps had rules, including an even larger number of scanners.

To conclude this section, the results of the association rule mining of VirusTotal scanners correctly deeming malicious apps in the *Hand-Labeled* and *Hand-Labeled 2019* datasets suggest the following. It seems that are a few scanners that are usually associated with correctly labeling malicious apps in both datasets as malicious. However, the number of those scanners is not large enough to fall within the range of three to six. This range of scanners is achieved through other scanners that occasionally appear in the scan reports of malicious apps and correctly label them as malicious. So, why do these scanners occasionally appear in the scan reports and not persist along with the ESET-NOD32 and SymantecMobileInsight scanners to form a larger set of correct and persistent scanners?

4.3. Sensitivity to VirusTotal's Dynamicity

In Section 1.2.1, we gave an example of VirusTotal's dynamicity and argued its potential impact on the performance of threshold-based labeling strategies: changing the optimal values of thresholds that help threshold-based strategies to assign labels to apps that accurately reflect their ground truths. In the previous section, despite finding that a range of thresholds between three and six yields the best MCC scores on the *Hand-Labeled* and *Hand-Labeled 2019* datasets, we noticed that the performance of labeling strategies utilizing these thresholds fluctuates at different points in time especially against the latter dataset. For example, as seen in Figure 4.3, the MCC scores of the vt$\geq$4, vt$\geq$5, and vt$\geq$6 labeling strategies fluctuate between 0.89 and 0.76 between September 27th, 2019 and November 8th, 2019. Moreover, using association rule mining, we found that the set of VirusTotal scanners that correctly deem malicious apps in both datasets as malicious appears to change with the exception of a few scanners. In this section, we investigate whether the reason behind such a fluctuation is indeed the dynamicity of VirusTotal.

Firstly, to check whether the threshold range of three to five scanners still helps threshold-based labeling strategies to accurately reflect the ground truths of apps in the *Hand-Labeled* and *Hand-Labeled 2019* datasets, we calculated the MCC scores achieved by threshold-based labeling strategies using thresholds between two and nine VirusTotal scanners based on scan reports downloaded on November 8th, 2019. Based on the plot in Figure 4.4, we found that the optimal threshold range of three-to-five scanners found on September 27th, 2019 ceased to exist for the following reasons. First, on the *Hand-Labeled* dataset, the performance of vt$\geq$3 increased to match that of vt$\geq$4, whereas vt$\geq$5 maintained the same MCC score. Second, the performance of threshold-based labeling strategies on the *Hand-Labeled 2019* dataset decreased gradually in two stages. After a steady performance with thresholds two



(a) *Hand-Labeled* (b) *Hand-Labeled 2019*

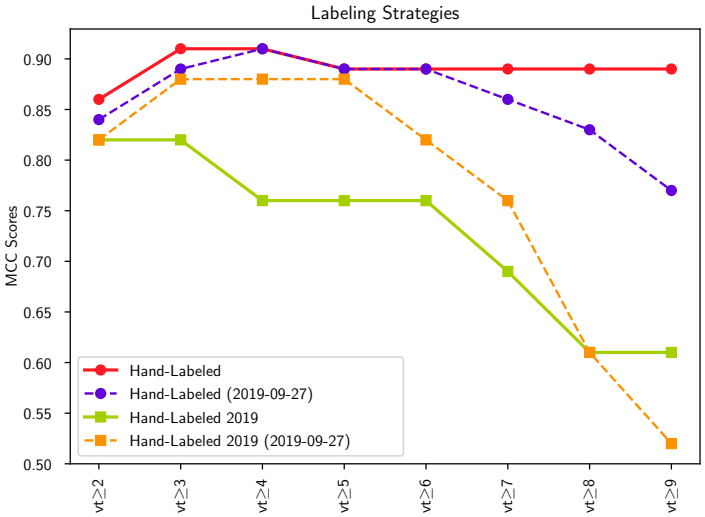
Figure 4.3.: The labeling accuracy of threshold-based labeling strategies using thresholds between three and six scanners against apps in *Hand-Labeled* and *Hand-Labeled 2019* datasets based on their `VirusTotal` scan reports downloaded between July 5^{th}, 2019 and November 8^{th}, 2019. Accuracy is calculated in terms of the MCC of each labeling strategy.

and three, the performance of threshold-based labeling strategies drops with $\texttt{vt}\geq 4$ to 0.76 MCC score. With a threshold of seven `VirusTotal` scanners, the performance drops again to an MCC score of 0.61 and stabilizes for $\texttt{vt}\geq 8$ and $\texttt{vt}\geq 9$. That makes the new range of optimal thresholds for apps in the *Hand-Labeled 2019* dataset to be between two and three scanners. Consequently, the intersection of the best-performing thresholds on both datasets yields a new range of optimal thresholds, namely three `VirusTotal` scanners.

While the performance of threshold-based labeling strategies on both datasets contributed to the change of the optimal range, the performance of such strategies on the *Hand-Labeled 2019* had the most impact. So, in understanding the reasons behind the change in the optimal range of thresholds, we focus on the performance of threshold-based labeling strategies on the *Hand-Labeled 2019* dataset.

We start by identifying which type of apps (i.e., malicious or benign), caused the change by examining their specificity ($\frac{TN}{N}$) and recall ($\frac{TP}{P}$) scores instead of the MCC score. We found that the performance of $\texttt{vt}\geq 3$, $\texttt{vt}\geq 4$, and $\texttt{vt}\geq 5$ on benign apps in the *Hand-Labeled 2019* dataset **did not change**; that is these labeling strategies correctly labeled the same benign apps at both points in time. Nonetheless, we found that these labeling strategies respectively had recall scores of 0.7, 0.6, and 0.6 on November 8^{th}, 2019 instead of 0.8, 0.8, and 0.8 on September 27^{th}, 2019. Since the total number of malicious apps in this dataset is ten, we can investigate the differences in the *positives* values in their scan reports and the different `VirusTotal` scanners that deemed them malicious on both dates. In Table 4.1, we detail the change in the *positives* values in terms of the `VirusTotal` scanners that deemed

Figure 4.4.: A comparison of the MCC scores achieved by threshold-based labeling strategies with thresholds between *two* and *nine* scanners on the *Hand-Labeled* and *Hand-Labeled 2019* as per `VirusTotal` scan reports downloaded on September 27th, 2019 and on November 8th, 2019.



the apps as malicious which (a) were added to the scan reports on November 8th, 2019, (b) were removed from the September 27th, 2019 scan reports, and (c) changed their verdicts between both dates.

Three apps out of ten did not encounter any change in their *positives* values. Nonetheless, one[3] of these apps maintained the same value of *positives* because two `VirusTotal` scanners changed their verdicts, namely `Zillya` changed its verdict from benign to malicious and `Trustlook` changed its verdict vice versa. We looked up the versions of these two scanners and found that the version of `Trustlook` remained at 1.0 between September 27th, 2019 and November 8th, 2019, whereas that of `Zillya` slightly changed from 2.0.0.3911 to 2.0.0.3946, which indicates an update. In fact, we found that `VirusTotal` only updated the versions of `Zillya` and `K7GW` and kept the same versions of all other scanners. So, the change in verdicts, we presume, is due to a technical decision made by firms maintaining those two scanners that were propagated to the scanners via new signatures databases.

However, `VirusTotal` contributed to altering the performance of `vt`$\geq$`3`, `vt`$\geq$`4`, and `vt`$\geq$`5` by removing scanners that correctly deemed two apps malicious from their scan reports. We use `VirusTotal`'s API to download the scan reports of apps; in this context, a *removal* of a scanner verdict means that we could not find its verdict in the downloaded

[3] 7658f70ae6acccfa9f3e900f8ae689603cc19d0b

Table 4.1.: The evolution of *positives* for apps in the *Hand-Labeled 2019* dataset that we deemed malicious after manual analysis and a detailed view of the VirusTotal scanners that were added/removed or changed their verdicts between September 27th, 2019 and November 8th, 2019 and how that affected the performance of vt≥3, vt≥4, and vt≥5. The check mark (✓) depicts whether the threshold-based labeling strategy managed to detect the malicious app.

App's SHA1 Hash	*positives* (September 27th, 2019)	*positives* (November 8th, 2019)	Added Positives	Removed Positives	Flipped to Positive	Flipped to Negative	vt≥3	vt≥4	vt≥5
8a9...	0	0	–	–	–	–			
bd9...	5	3	–	ESET-NOD32 Fortinet Ikarus	Cyren	–	✓		
765...	7	7	–	–	Zillya	Trustlook	✓	✓	✓
5be...	12	13	Ikarus	–	–	–	✓	✓	✓
6da...	7	6	–	–	–	Trustlook	✓	✓	✓
c70...	13	13	–	–	Symantec	Zoner	✓	✓	✓
b9f...	10	12	Ikarus	–	AegisLab K7GW	McAfee	✓	✓	✓
a0a...	8	11	Fortinet	–	Zillya Zoner	–	✓	✓	✓
90e...	6	1	–	ESET-NOD32 Fortinet Ikarus Yandex	–	Cyren			
0d5...	1	1	–	–	–	–			

scan report under the *scans* attribute. On the GUI interface of VirusTotal, this is usually displayed as a gray "Unknown" next to the scanner's name. In particular, the scanners ESET-NOD32, Fortinet, and Ikarus were removed from one[4] app's scan report, effectively reducing its *positives* value from five on September 27th, 2019 to only three on November 8th, 2019. This change prevented the vt≥4 and vt≥5 labeling strategies from correctly deeming the app as malicious. The same scanners along with Yandex were not included in the second[5] app's November 8th, 2019 scan report, which brought the value of *positives* from six to one, putting the app beyond the reach of all of the three aforementioned strategies.

Unfortunately, we cannot identify the reasons behind VirusTotal's decision to alter the set of scanners it includes in an app's scan report. It is a puzzling fact that VirusTotal added the Ikarus and Fortinet scanners to the scan reports of some apps, whilst removing them from the reports of others. The more confusing fact is that the removed scanners contain the ESET-NOD32 scanner, which we found to have correctly labeled

[4] bd97c85d38bd5bfc5e29b05b1a3a81b12949065a
[5] 90e6ac481fdd497f152234f1cd5bec6d40f50037

the apps as malicious on September 27th, 2019. This seemingly haphazard inclusion and exclusion of scanners within periods as small as two weeks contribute to the frequent change of the currently optimal range of thresholds. While such change does not impact older apps, such as the ones in the *Hand-Labeled* dataset, its impact is noticeable on newer apps, such as the ones in the *Hand-Labeled 2019* dataset. With this set of measurements, we unveil the first limitation of `VirusTotal`, which has a direct impact on the performance of threshold-based labeling strategies, viz.:

`VirusTotal` Limitation 1

`VirusTotal` frequently changes the set of scanners it includes in the scan reports of apps over time by including and excluding the verdicts of scanners regardless of the quality of those verdicts. In some cases, particularly with newly-developed malicious apps, this frequent change causes the performance of threshold-based labeling strategies that accurately labeled those apps as malicious at one point in time to fluctuate over time. Effectively, this frequent change undermines the utilization of fixed thresholds.

4.4. Finding the Optimal Threshold

In the previous section, we found that `VirusTotal` changes the set and versions of scanners it includes in the scan reports of apps. This change impacts the long-term labeling accuracy of threshold-based labeling strategies. For example, between September 27th, 2019 and November 8th, 2019, `VirusTotal`'s dynamicity caused the decrease of $\texttt{vt}\geq 4$'s MCC score from 0.89 to 0.76 (i.e., a decrease of 14.61%). However, we also noticed that during this period the labeling strategies $\texttt{vt}\geq 2$ and $\texttt{vt}\geq 3$ maintained their MCC scores, and that of $\texttt{vt}\geq 1$ had an increase of MCC score. This suggests that the previously-discussed aspects of `VirusTotal`'s dynamicity cause threshold-based labeling strategies to trade places in terms of the most accurate ones. In other words, at any given moment in time, a (different) subset of threshold-based labeling strategies will depict the most accurate labeling strategies (i.e., optimal thresholds). So, researchers can use a reference set of Android apps whose ground truth is known, download their latest `VirusTotal` scan reports, compare the labeling accuracy of all thresholds between one and 60 and choose the threshold that yields the best score. In this section, we investigate the feasibility of this brute force approach to identify the optimal thresholds of `VirusTotal` scanners at any point in time.

Algorithm 1 depicts a simple algorithm to find the current threshold of `VirusTotal` scanners that yields the most accurate labels. To assess the quality of labels given by different threshold-based labeling strategies, this algorithm requires the presence of a dataset (A) of pre-labeled Android malicious and benign apps. As mentioned earlier in this paper, the most reliable ground truth (γ) can be generated using manual analysis and app labeling. Without such a reliable ground truth (γ) that acts as a reference to compare

Algorithm 1 An algorithm to find the current optimal threshold of `VirusTotal` scanners to use in labeling Android apps.

```
1:  procedure FindCurrentOptimalThreshold(A, γ)
2:      tmpResults = {}
3:      for all α ∈ A do
4:          truth = γ_α
5:          response = VirusTotal.rescanApp(α)
6:          if response == True then
7:              report = VirusTotal.downloadReport(α)
8:              if report != Null then
9:                  positives_α = report["positives"]
10:                 for all τ ∈ {1, 2, 3, ..., 60} do
11:                     if positives_α ≥ τ then
12:                         label_α = malicious
13:                     else
14:                         label_α = benign
15:                     tmpResults[vt≥ τ].append(label_α)
16:
17:     bestThreshold = ""
18:     bestScore = 0.0
19:     for all τ ∈ {1, 2, 3, ..., 60} do
20:         currentScore = calculateScore(tmpResults[vt≥ τ], γ)
21:         if currentScore ≥ bestScore then
22:             bestScore = currentScore
23:             bestThreshold = vt≥ τ
        return bestThreshold, bestScore
```

against, one has to choose a subjective threshold (τ) that one believes represents the nature of apps in (A) and use it as ground truth (γ_τ) If so, the only threshold that would generate labels mimicking (γ_τ) would be (τ) itself. Effectively, we wind up with the exact problem of choosing subjective thresholds based on personal views, as discussed in Section 4.1.

Relying on manual analysis already introduces infeasibility to the algorithm. However, assuming the existence of pre-labeled apps, another problem arises. As discussed earlier, the immaturity of newly-developed Android malware and the dynamicity of `VirusTotal` lowers the values of the *positives* attribute in the scan reports of those apps, which, in turn, lowers the thresholds needed to label them accurately. Without access to newly-developed Android malware, researchers risk choosing thresholds based on the scan reports of old malicious apps, which are much higher than the thresholds required to detect new malware. For example, in Figure 4.1, if a researcher only has access to the *Hand-Labeled* dataset, on October 11^{th}, 2019, they might opt to use the `drebin` labeling strategy because it exhibits stable performance of high MCC scores. However, this labeling strategy will perform much worse on newer apps in the *Hand-Labeled 2019* dataset (i.e., it does not generalize to newer malicious apps). Consequently, researchers adopting this brute force approach to finding the currently optimal thresholds need to continuously update their reference datasets with newly-developed and discovered Android (malicious) apps.

If the reference dataset (A) satisfies the previous conditions of being diverse, regularly updated, and accurately labeled, then identifying the currently optimal threshold can be performed as follows. For each app in the dataset ($\forall \alpha \in A$), the latest scan report of (α) needs to be acquired. Firstly, the user has to issue a re-scan request to `VirusTotal`, which takes around five minutes to complete (line `5`). As discussed earlier, this request can be issued using the platform's web interface or using the API interface. In general, under the academic license, a total of 20K requests can be issued per day. So, depending on the size of the reference dataset (A), the process of rescanning all apps might take days. Furthermore, we recently were forbidden from issuing this type of request using our academic license. As of the date of writing this thesis, we are unaware of whether `VirusTotal` prevents academic licenses from issuing this type of request or whether we are encountering an individual technical difficulty. We consider the decision of `VirusTotal` not to automatically and regularly rescan apps as another limitation of the platform:

`VirusTotal` Limitation 2

`VirusTotal` does not rescan the apps it possesses on a regular basis and delegates this task to manual requests issued by its users. One direct consequence of this decision is prolonging the process of acquiring up-to-date scan reports of apps. Furthermore, in our experience, `VirusTotal` does not allow issuing rescan requests using academic licenses.

In line `7`, after the rescan requests are completed, researchers need to download the up-to-date scan reports from `VirusTotal`. Similar to the rescan API requests, download

requests are limited to 20K requests per day, which might add a few more days to the process. Between lines 8 and 23, the process becomes straightforward. Using thresholds (τ) between one and 60, the labels of apps in (A) are calculated and stored in a temporary structure under the key vt$\geq \tau$ (line 15). The stored labels are then compared against the ground truth (γ), and a score is calculated, say MCC. The threshold-based strategy that yields the best score is returned to the user as the currently optimal one.

To sum up, in theory, the process of finding the currently optimal threshold of scanners by comparing the performance of all possible thresholds is straightforward. However, in order for this process to be effective, a reference dataset of manually pre-labeled apps needs to be at the researchers' disposal. Moreover, given the dynamicity of VirusTotal and the immaturity of the scan reports of newly-developed Android (malicious) apps forces researchers to keep such a reference dataset up-to-date, which is not feasible given the design of VirusTotal, which delegates the re-scanning of apps to its users.

4.5. Summary

In this chapter, we focused on the most intuitive labeling strategies used to interpret VirusTotal's scan reports into labels, namely threshold-based labeling strategies. Surveying the literature, it seems that researchers choose fixed thresholds based on their understanding of what is malicious. Nevertheless, our measurements show that only a subset of thresholds should be used to label Android apps. Furthermore, we found that the older the apps, the more mature their VirusTotal scan reports, which is reflected in the *positives* values being high enough to enable different threshold-based labeling strategies to label apps as malicious accurately, which will help us address **(RQ2)**. Our measurements revealed one aspect of VirusTotal's dynamicity **(RQ5)**: the platform seems to frequently and randomly alter the set of scanners it includes in an app's scan report, which sometimes causes the removal of scanner verdicts that correctly labeled the apps. This change negatively affects the ability of threshold-based labeling strategies to recognize the malignancy of apps, particularly newer apps whose scan reports have fewer scanners that deem them as malicious. Furthermore, the frequency of such a change prevents researchers from using fixed thresholds to label apps, effectively forcing them to identify the currently optimal threshold before labeling Android apps. As part of our solution to address **(RQ4)**, we devised an algorithm to identify the currently optimal threshold by brute-forcing all possible values of thresholds, viz. between one and 60. Despite its theoretical simplicity, identifying the currently optimal threshold by trying all possible thresholds faces a number of limitations. Firstly, researchers need access to a reference dataset of (manually) pre-labeled Android apps, which need to be frequently updated in order to find thresholds that generalize to newly-developed Android (malicious) apps. Secondly, VirusTotal's design decision to delegate the re-scanning apps to users significantly prolongs the process of finding the optimal threshold, provided that it is still available under academic licenses. Consequently, there is a need for labeling strategies that can find an optimal method to

interpret VirusTotal's scan reports without the need for the regular re-scan and reanalysis of such scan reports.

5. Maat: A Framework to Optimally Utilize VirusTotal

This chapter presents the main contribution of this thesis: a systematic method, called Maat*, that automatically analyzes* `VirusTotal` *scan reports of pre-labeled apps to identify the set of correct and stable* `VirusTotal` *scanners and uses such information to build ML-based labeling strategies. We describe how the set of correct and stable scanners is identified along with insights about* `VirusTotal` *and its scanners that we made in this process. Parts of this chapter have previously appeared in peer-reviewed publications [9] and [116], co-authored by the author of this thesis.*

In the previous chapter, we discussed the subjectivity of choosing the optimal thresholds to label Android apps based on their `VirusTotal` scan reports. Using subjective threshold-based labeling strategies to label apps might yield inaccurate labels that negatively impact the performance of malware detection methods and hinder the comparison of malware detection approaches that use different thresholds to label apps. However, we found that there is a subset of thresholds that can yield more accurate labels than others. Unfortunately, given the dynamicity of `VirusTotal`, researchers have to identify the current optimal thresholds, which entails a semi-automatic process that is infeasible to perform on a regular basis. In summary, threshold-based labeling strategies suffer from the problems of subjectivity, sensitivity to `VirusTotal`'s dynamicity, and the infeasibility of brute-forcing the current optimal thresholds.

Our framework, Maat, is designed to address these problems as follows. First, to avoid the subjectivity of threshold-based labeling strategies, Maat provides the research community with a systematic and standardized method to analyze `VirusTotal` scan reports to devise labeling strategies. Second, Maat's processes of analyzing `VirusTotal` scan reports and of devising ML-based labeling strategies are on-demand and fully automated. Third, the resulting ML-based labeling strategies do not rely on a fixed number of `VirusTotal` scanners to label apps as malicious and benign. Instead, using two different types of features extracted from `VirusTotal` scan reports, Maat relies on ML algorithms to identify the currently correct and stable `VirusTotal` scanners, which makes them less susceptible to the dynamicity of `VirusTotal` and the changes it introduces to apps' scan reports.

In this chapter, we discuss the process that Maat adopts to extract features (Section 5.5) from `VirusTotal` scan reports to train ML-based labeling strategies. We first give an overview of how such labeling strategies are built (Section 5.1), then detail how certain

features are extracted from scan reports in Section 5.2, and Section 5.3. Lastly, we enumerate the types of features extracted from the scan report of Maat's training dataset (Section 5.5). The results of our measurements helped decide upon the features to extract from VirusTotal scan reports to train ML-based labeling strategies that we evaluate in Chapter 6.

5.1. Overview

Prior to delving into the measurements and experiments we performed, we briefly discuss how our framework, Maat, analyzes VirusTotal scan reports to build ML-based labeling strategies. As seen in Figure 5.1, Maat starts by analyzing the VirusTotal scan reports of apps in the training dataset that are reanalyzed via VirusTotal and downloaded at different points in time (i.e., $t_0, t_1, ..., t_m$). The training dataset that Maat uses to train ML-based labeling strategies comprises the scan reports of apps in the *AMD+GPlay* dataset gathered between November 2018 and November 8^{th}, 2019.

In phase (1) from Figure 5.1 we identify the VirusTotal scanners that achieve an average overall correctness score[1] of at least 90% between November 2018 and November 8^{th}, 2019 as the most correct scanners. Maat also finds the scanners that changed their verdicts at most 10% of the time (i.e., were stable 90% of the time). The output of this phase is an intersection of the most correct and stable VirusTotal scanners. These scanners are considered in the next phase to extract features from the scan reports. The exact processes we adopted to find the most correct and stable scanners are detailed in Section 5.2 and Section 5.3, respectively.

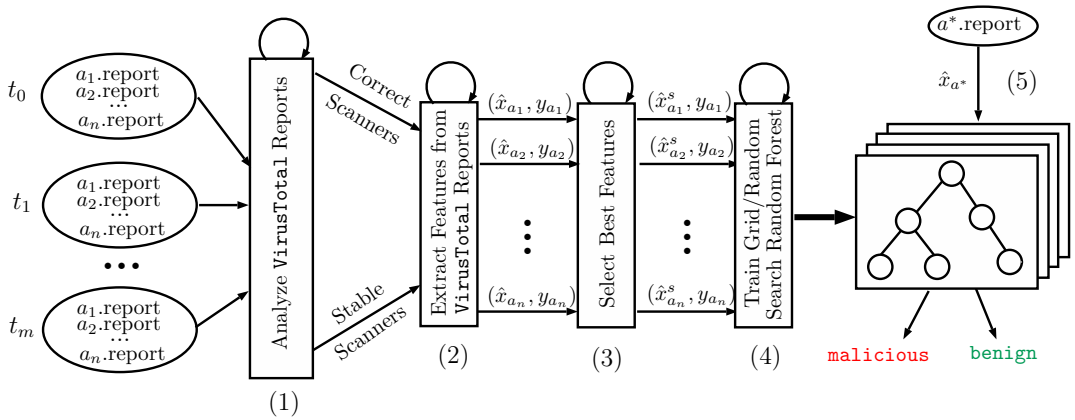
In phase (2), we extract features from the VirusTotal scan reports of apps in the *AMD+GPlay* dataset. That is, each app in the dataset will be represented as a vector of numerical features extracted from its most recent VirusTotal scan report. There are two types of features we extract from the reports, namely *engineered* features and *naive* features.

Engineered features attempt to leverage the insights we gained from the previous sections (e.g., which scanners are correct). Firstly, based on the output from phase (1), we consider the verdicts given to apps in the training dataset only by the set of most correct and stable scanners. To accommodate the impact of time on the maturity of an app's scan report, we also include the age of a scan report in years, the number of times an app has been submitted for (re)analysis (i.e., *times_submitted*), the *positives* attribute, and the *total* attribute in this feature set. Lastly, to capture any patterns that Android (malicious) apps share in terms of functionalities and runtime behaviors, we extract from the VirusTotal scan reports the permissions that apps request in their AndroidManifest.xml files, and the tags given to them by VirusTotal (e.g., *checks-gps*, *contains-elf*, *sends-sms*, etc.).

Naive features do not consider the outputs of phase (1). With naive features, we consider **only** the verdicts given by **all** VirusTotal scanners to the apps in the training dataset. So,

[1]The correctness score is based on Mohaisen's definition [92] and calculated against the ground truths given by the authors of *AMD* [153] to apps using their hybrid analysis process that combines reliance on VirusTotal scan reports and manual analysis as discussed in Section 1.3.4.

Figure 5.1.: The process adopted by Maat to construct ML-based labeling strategies by analyzing VirusTotal scan reports and training a random forest to label apps as malicious and benign according to their VirusTotal scan reports.



the feature vector extracted from a VirusTotal scan report will be a sequence of integers depicting the labels given by each scanner to an app (i.e., -1 for not scanned, 0 for scanned and deemed benign, and 1 for scanned and deemed malicious). For example, assume that the scan report of an arbitrary app (α^*) contained scan results of three scanners, that respectively deemed (α^*) as malicious, malicious, and benign; the feature vector depicting this scan report will be ($\hat{x}_{\alpha^*} = (1, 1, 0)$). With naive features, we allow ML-based labeling strategies to utilize the verdicts of all VirusTotal scanners regardless of their correctness or stability.

Phase (3) is an **optional** phase that selects the most informative features extracted from the training dataset's VirusTotal scan reports, so that the decision trees that use the *max_depth* parameter can randomly select features from a set of more informative ones. To avoid having to choose the number of features to select arbitrarily, we utilize the SelectFromModel [126] technique to select the most informative features automatically. In essence, this technique selects features based on the importance given to them by a model (e.g., logistic regression, support vector machine, decision trees, etc.). For example, during training, decision trees iteratively utilize a criterion (e.g., Gini index), to decide upon the next feature to consider in splitting data points into two, or more, classes; in our case, this feature could be a scanner's verdict regarding the label of an app. Ultimately, the trained tree will compile a set of features that it used during splitting and assign an *importance* value to each one of them. The SelectFromModel feature selection technique uses such importance values and returns to the user those features with importance values more than a preset threshold (i.e., 1×10^{-5} in the case of decision trees). For our experiments, we rely on decision trees as the model used by the SelectFromModel technique to extract the most informative

features.

We envision the process of utilizing the features extracted from `VirusTotal` scan reports to label apps as a series or combination of questions, such as *how many scanners deem the app malicious? how old is the app? does a renowned scanner (e.g., `AVG`) deem the app as malicious?* The machine learning classifier that mimics this model, we reckon, is a decision tree. In order not to rely on the decisions made by a single tree, Maat trains ML-based a labeling strategy as a collection of trees or a *random forest* using the framework `scikit-learn`. To estimate the hyperparameters (e.g., the maximum depth each tree is allowed to grow), that train the most effective forests, we use the technique of *grid search* [127] to select from among a set of parameters listed in Appendix E.

The output of phase (4) is a random forest that takes a vector of numerical features extracted from an app's `VirusTotal` scan report and returns a label depicting the class of the app (i.e., 1.0 for malicious and 0.0 for benign). Effectively, this random forest is a labeling strategy. In phase (5), given the `VirusTotal` scan report of an arbitrary Android app, the report is represented as a feature vector that matches the features used by the random forest (e.g., naive versus engineered features), and is used to predict the app's class.

5.2. Correctness of `VirusTotal` Scanners

The results in Table 2.1 imply that some `VirusTotal` scanners fail to recognize the malignancy of some malicious apps and that some are more correct than others. As part of extracting engineered features to train ML-based labeling strategies, in this section, we describe the process adopted by Maat to find the set of the most accurate `VirusTotal` scanners during a given period of time. Given the scan reports of apps in the training dataset gathered over a period of time, Maat builds on Mohaisen et al.'s definition of scanner *correctness* to identify the set of correct `VirusTotal` scanners over this period. In [92], Mohaisen et al. defined correctness as follows: For a given dataset, the correctness of an antiviral scanner is the number of correct detections normalized by the size of the dataset. We extend this definition in two manners. First, we include benign apps in the calculation of the correctness score. Second, we extend the correctness score to be based on other metrics apart from the accuracy metric (i.e., $\frac{TP+TN}{P+N}$), including the MCC score, recall ($\frac{TP}{P}$), and the F_1 score. In this thesis, we rely on the accuracy metric in calculating the correctness of `VirusTotal` scanners.

For each `VirusTotal` scanner, Maat calculates the number of apps in the *AMD+GPlay* dataset that it managed to correctly label as malicious and benign and divides that number by the size of the dataset. The correctness scores of each scanner at various points in time are then averaged. Scanners whose correctness scores are greater than or equal to 0.90 are included in the set of correct `VirusTotal` scanners. We chose the number 0.90 to tolerate any fluctuations in the correctness rate that may occur, for example, due to changing policies on how to label ambiguous malware types, such as `Adware`, or the temporary use of inadequate versions of scanners. So, the definition of a correct scanner we adopt in this

thesis is:

Definition

A correct `VirusTotal` scanner is one that achieves a correctness score greater than or equal to a value between 0.0 and 1.0 (default: 0.90) on any given dataset of pre-labeled apps. The score is defined in terms of a metric, such as accuracy, MCC, recall, F_1 score, etc., and is meant to depict the ability of the `VirusTotal` scanner to correctly assign labels to apps in the dataset that reflect their ground truth.

Using a value of 0.90 and the accuracy metric, we retrieved as list of 18 scanners that Maat identified to be correct between November 2018 and July 5^{th}, 2019 on the *AMD+GPlay* dataset, namely `AhnLab-V3`, `Avira`, `Babable`, `CAT-QuickHeal`, `Comodo`, `Cyren`, `DrWeb`, `ESET-NOD32`, `F-Secure`, `Fortinet`, `Ikarus`, `K7GW`, `MAX`, `McAfee`, `NANO-Antivirus`, `Sophos`, `SymantecMobileInsight`, and `Trustlook`. In Section 5.6, we discuss why we chose this time period to extract the set of correct scanners. This list noticeably differs from the ten scanners Arp et al. used in 2014 to label apps in their *Drebin* dataset [15]; only five *Drebin* out of ten are included in Maat's list of correct scanners. Apart from deeming them as popular and trustworthy, the authors did not mention why they chose those ten scanners. Intuitively, Arp et al. used a different set of Android apps and older versions of scan reports, which may imply that the set of correct scanners might change from one set of Android apps to another and from time to time. So, we hypothesize that there are no set of `VirusTotal` scanners that are universally correct on different datasets. To find support for or against this hypothesis, we retrieved the set of scanners that maintained correctness scores of 90% or higher over a period of time for the *AMD*, *AMD+GPlay*, *GPlay*, *Hand-Labeled* and *Hand-Labeled 2019* datasets (i.e., the datasets with known ground truths).

Since we downloaded the apps in the *Hand-Labeled* and *Hand-Labeled 2019* datasets after downloading apps in the *AMD+GPlay* dataset, we started re-scanning and downloading their latest `VirusTotal` scan reports at later points in time, viz. starting from July 5^{th}, 2019 instead of November 2018. So, to perform an objective measurement, we retrieved the list of correct `VirusTotal` scanners for the *AMD+GPlay*, *Hand-Labeled*, *Hand-Labeled 2019* datasets within the same time period, namely between July 5^{th}, 2019 and November 8^{th}, 2019. As seen in Table 5.1, the set of `VirusTotal` scanners that have correctness scores of at least 0.90 differ from one dataset to another, despite being based on scan reports downloaded within the same period. Only a set of five scanners is shared by all datasets, viz. `ESET-NOD32`, `Fortinet`, `Ikarus`, `McAfee`, and `SymantecMobileInsight`.

The results in Table 5.1 suggest that the set of `VirusTotal` scanners that are correct over time might change depending on (a) the time period within which the `VirusTotal` scan reports were gathered, and (b) the dataset itself and its composition in terms of benign and malicious apps. In Chapter 4, we revealed one aspect of `VirusTotal`'s dynamicity, namely that it frequently changes the set of scanners included in the scan reports of apps.

Table 5.1.: The set of VirusTotal scanners that had accuracy-based correctness scores of at least 0.90 between July 5^{th}, 2019 and November 8^{th}, 2019. Emboldened scanners depict the intersection of the sets of correct scanners of the four datasets.

Dataset	Scanner(s)	Total
AMD+GPlay (49.06% benign + 50.04% malicious)	AhnLab-V3, Avira, CAT-QuickHeal, Comodo, Cyren, DrWeb, **ESET-NOD32**, F-Secure, **Fortinet**, **Ikarus**, K7GW, MAX, **McAfee**, NANO-Antivirus, Sophos, **SymantecMobileInsight**, TheHacker, Trustlook	18
AMD (malicious)	Avira, CAT-QuickHeal, DrWeb, **ESET-NOD32**, F-Secure, **Fortinet**, **Ikarus**, MAX, **McAfee**, NANO-Antivirus, Sophos, **SymantecMobileInsight**	12
Hand-Labeled (76% benign + 24% malicious)	AhnLab-V3, CAT-QuickHeal, Cyren, **ESET-NOD32**, **Fortinet**, **Ikarus**, K7GW, **McAfee**, Sophos, **SymantecMobileInsight**, Trustlook	11
Hand-Labeled 2019 (90% benign + 10% malicious)	Ad-Aware, AegisLab, AhnLab-V3, Alibaba, Arcabit, Avast-Mobile, BitDefender, ClamAV, Cyren, DrWeb, **ESET-NOD32**, Emsisoft, F-Secure, FireEye, **Fortinet**, GData, **Ikarus**, Jiangmin, K7AntiVirus, K7GW, Kaspersky, Kingsoft, MAX, MalwareBytes, **McAfee**, McAfee-GW-Edition, MicroWorld-eScan, Microsoft, NANO-Antivirus, Qihoo-360, SUPERAntiSpyware, **SymantecMobileInsight**, Trustlook, ViRobot, Yandex, ZoneAlarm, Zoner	37

Consequently, in calculating the correctness of a VirusTotal scanner, we only consider the VirusTotal scan reports in which the scanner is included as a measure of fairness to the scanner. The decision to only focus on scan reports in which the verdict of the VirusTotal scanner under study is included impacts metrics that do focus on the total number of scanned malicious (P) and benign (N) apps, such as accuracy ($\frac{TP+TN}{P+N}$), recall ($\frac{TP}{P}$), and specificity or negative recall ($\frac{TN}{N}$). Metrics that do not consider the (P) and (N) values, such as MCC, are not going to be affected by this decision. Despite that decision we made, VirusTotal's changing of the set of scanners it includes in the scan reports of apps can still impact the correctness of a scanner as follows. Say that a scanner (σ) was included in 90 VirusTotal scan reports of a dataset of 100 Android apps. Moreover, consider that (σ) managed to correctly classify all of those 90 apps, earning it a correctness score using the accuracy metric of 0.90. If VirusTotal removes the verdicts of (σ) from the scan reports of 20 apps, depending on which verdicts were removed, the accuracy-based correctness of (σ) would be between 80 out of 80 (i.e., 1.0) and 70 out of 80 (i.e., 0.87), which is already enough to exclude (σ) from the set of correct scanners for this dataset of 100 apps. One can notice this impact in Table 5.1 as the set of correct VirusTotal scanners extracted from the *AMD+GPlay* dataset using VirusTotal scan reports downloaded between November 2018 and July 5^{th}, 2019 slightly differs from the set extracted from the same dataset between July 5^{th}, 2019 and November 8^{th}, 2019. In particular, the scanner Babable was excluded from the set of correct scanners in the latter period. Upon further investigation, we found that the verdicts of Babable were, on average, included in 77% of the VirusTotal scan reports of apps in the *AMD+GPlay* dataset between November 2018 and July 5^{th}, 2019 as

opposed to only 18.6% between July 5^{th}, 2019 and November 8^{th}, 2019, which brought down its correctness from slightly above 0.90 to 0.59.

As for the composition of the dataset, we noticed that the larger the number of apps pre-labeled as benign in a dataset, the bigger the set of correct scanners we can retrieve from it. Another way of looking at this is that the larger the number of apps pre-labeled as malicious in a dataset, the smaller the set of correct scanners we can retrieve from it. In theory, scanners should be assessed equally based on their ability to correctly label malicious and benign apps. That is, true positives are not more worthy than true negatives. In this context, we used the balanced dataset *AMD+GPlay* of malicious and benign apps to find the set of `VirusTotal` scanners that accurately label both classes of apps. We also attempted to understand whether there is a correlation between the composition of a dataset and the number of correct scanners we can retrieve from within a given period. Mislabeling a benign app as malicious (i.e., false positives) is known to have more of a negative effect on the reputation and, hence, antiviral scanners' popularity than mislabeling a malicious app as benign [112]. Consequently, antiviral scanners are said to be reluctant to label apps as malicious and opt for assigning a label of benign to apps by default. This reluctance makes it difficult to assess whether a benign label given by an antiviral scanner to an app is a result of analyzing the apps and consciously labeling it as benign or just a default label. Labeling an app as malicious, however, may imply a thorough analysis process that led to deeming an app as such, especially given the negative impacts of false positives. So, we speculate that benign apps alone do not reveal the detection ability of an antiviral scanner. Similar to the problem of defining malignancy that we discussed in Section 2.5, we cannot assert why and how antiviral scanners deem apps as malicious or benign without access to their internal analysis and detection processes.

We extract the list of correct `VirusTotal` scanners as part of Maat's process to train ML-based labeling strategies. However, one of the research questions postulated in this thesis was to check whether there is a universal set of `VirusTotal` scanners that are more correct than others over time and across datasets (i.e., **(RQ3)**). The answer depends on the definition of a universal set. If researchers expect that set of scanners to maintain the same scanners, then the answer is no. The combination of `VirusTotal`'s dynamicity and the composition of a dataset in terms of benign and malicious apps causes this set to change on a regular basis. However, in our measurements, we noticed that there is a subset of scanners that persist across different datasets, viz. the five scanners `ESET-NOD32`, `Fortinet`, `Ikarus`, `McAfee`, and `SymantecMobileInsight`. For these datasets during that particular period of time, this set of five scanners can be considered as a universal set of (correct) scanners. To conclude, unless the *universality* of `VirusTotal` scanners is confined to particular datasets and time periods, given `VirusTotal`'s dynamicity, there is no point behind pursuing a set of `VirusTotal` scanners that persists across different datasets and time periods.

To further investigate the aspects of `VirusTotal`'s dynamicity, we used Maat's API to tabulate and visualize the correctness of `VirusTotal` scanners to provide its users with further insights about the performance of different scanners over time. For example,

Table 5.2.: The change in the correctness rates of the *Drebin* VirusTotal scanners on apps in the *AMD* dataset between November 2018 and November 8th, 2019, grouped by malware type.

Drebin Scanners / Malware Types	AVG	Avira	BitDefender	ClamAV	ESET	F-Secure	Kaspersky	McAfee	Panda	Sophos
Adware	*0.01→0.01*	0.91↗0.99	0.45↘0.0	0.10→0.10	0.98↗0.99	0.53↗0.99	*0.26↗0.33*	0.95↗0.99	0.0→0.0	0.92↗0.95
Backdoor	0.92↗0.93	0.96↗0.99	0.97↘0.1	0.61→0.61	0.99↗1.0	0.89↗0.99	0.98→0.98	0.99→0.99	0.01→0.01	0.98↗0.99
HackerTool	0.99↗1.0	0.99→0.99	**0.99↘0.90**	**0.91→0.91**	*0.81→0.81*	0.89↗1.0	*0.75→0.75*	0.99→0.99	0.32→0.32	0.99→0.99
Ransom	0.98↗0.99	0.99→1.0	0.98↘0.0	0.05↗0.06	0.99↗1.0	0.89↗1.0	0.99↗1.0	0.99↗1.0	0.0→0.0	0.99→0.99
Trojan	0.84→0.84	0.96↗1.0	0.88↘0.01	0.10↘0.07	0.96→0.96	0.75↗1.0	0.88→0.88	0.99↗1.0	0.0→0.0	0.97↗0.98
Trojan-Banker	0.98↗0.99	0.99→0.99	0.99↘0.0	0.09→0.09	0.99↗1.0	0.92↗1.0	0.99→0.99	0.99↗1.0	0.0→0.0	0.99↗1.0
Trojan-Clicker	0.96↗1.0	1.0→1.0	1.0↘0.0	0.32→0.32	1.0→1.0	0.87↗1.0	0.96↘0.90	1.0→1.0	0.0→0.0	1.0→1.0
Trojan-Dropper	0.98↗0.99	0.96↗0.99	0.96↘0.03	0.10↗0.11	0.97↗0.99	0.80↗0.99	0.97↗0.98	0.98↗1.0	0.02→0.0	0.97↗0.99
Trojan-SMS	0.88→0.88	0.99↗1.0	0.98↘0.0	0.24↘0.15	0.99→0.99	0.88↗1.0	0.98↗0.99	0.99↗1.0	0.0→0.0	0.99→0.99
Trojan-Spy	0.05↗0.87	0.99↗1.0	1.0↘0.0	0.01→0.01	0.99↗1.0	0.61→1.0	0.99→0.99	0.99↗1.0	0.0→0.0	0.99→0.99
Overall Correctness	0.34↗0.40	**0.94↗1.0**	0.68↘0.02	0.15↘0.13	**0.99↗1.0**	0.66↗**0.99**	0.57↗0.61	**0.97↗1.0**	0.01→0.01	**0.95↗0.97**

Figure 5.2 displays the correctness scores of ten scanners utilized by Arp et al. in [15] to label apps in the *Drebin* dataset. These scanners are AntiVir[2], AVG, BitDefender, ClamAV, ESET, F-Secure, Kaspersky, McAfee, Panda, and Sophos. We refer to this group of scanners as the *Drebin* scanners.

The results in Table 5.2 depict the evolution of the overall accuracy-based correctness of the *Drebin* scanners on the *AMD* dataset according to VirusTotal scan reports downloaded during November 2018 and on November 8th, 2019 along with the correctness per malware type. Focusing solely on the overall correctness rates, it is evident that some scanners are indeed more correct than others (e.g., ESET, McAfee, and Sophos). The following observations can be made upon considering the correctness rates per malware type. Firstly, scanners that have decent overall correctness usually score high correctness rates on all malware types. As for the correctness rates per malware type, on the one hand, some scanners achieve high correctness rates on the majority of malware types yet struggle with others. For example, AVG and Kaspersky appear to struggle with apps belonging to the Adware malware type (emphasized red). On the other hand, some scanners excel against particular malware types, such as BitDefender and ClamAV, which continue to correctly detect the HackerTool malware type. As seen in emboldened green, BitDefender and ClamAV appear to score decent correctness rates on the HackerTool malware type, whilst struggling with all other types. Lastly, some scanners, such as Panda, appear to struggle on all malware types, even though some apps in the *AMD* dataset are as old as 2010 [153].

The recorded performances of some scanners in Figure 5.2 do not coincide with the

[2]AntiVir refers to the antiviral scanner developed by Avira, which is the name that VirusTotal uses for this scanner now.

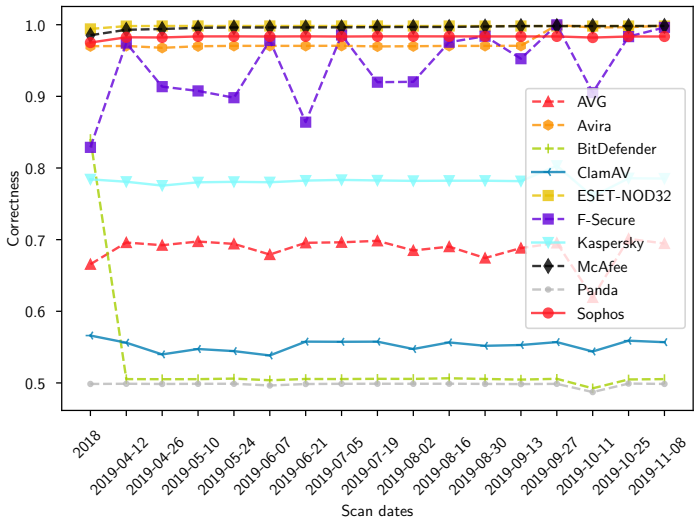


Figure 5.2.: The overall correctness rates of the *Drebin* scanners on apps in the *AMD+GPlay* dataset between November 2018 and November 8^{th}, 2019.

reputation of those scanners in the market. For example, despite its mediocre performance as reported by `VirusTotal`, `BitDefender` continues to be given good reviews by users on the Google Play marketplace and, more importantly, on platforms that assess the effectiveness of antiviral software such as *AV-Test* [59]. Given that `VirusTotal` states that the versions of scanners it uses *"may differ from commercial off-the-shelf products. The [antiviral software] company decides the particular settings with which the engine should run in VirusTotal"* [148], we compared the `VirusTotal` version of `BitDefender` against the one that can be found on the Google Play app marketplace. We found that, as of September 2019, the version used by `VirusTotal` for `BitDefender` is 7.2, whereas the versions available on Google Play have codes between 3.3 and 3.6. The 7.2 version of `BitDefender` corresponds to a free edition version developed for Windows-based malware that targets older versions of Windows such as `Windows XP` [83]. The positive reputation that `BitDefender` has in the market suggests that using its adequate version (i.e., the one that is designed to detect Android malware), would yield a detection performance better than the version on `VirusTotal`. To verify this, we downloaded and installed the latest version of the `BitDefender` scanner from the Google Play marketplace, installed it on an Android Virtual Device (AVD), and used it to scan ten apps randomly sampled from the *AMD* dataset. Unlike the results obtained from `VirusTotal` that the scanner is unable to detect any of those malicious apps, we found that `BitDefender` detects 70% of the sampled apps. Similar to the case of `BitDefender`, we found that `VirusTotal` uses a version of the `Panda` that is not dedicated to Android malware. So, we repeated the exact measurement above using the same ten malicious apps and the proper version of `Panda` and found

that such a version can detect 30% of the apps instead of none as per the version used by VirusTotal.

So, do some antiviral software companies believe that offering older versions on VirusTotal will encourage users to download their products instead of relying on the online version available on VirusTotal? Do such companies enforce a fee on VirusTotal in order to use their latest products, which the latter did not agree to? Answering those questions is not in the scope of this thesis and, in fact, impossible to answer on behalf of antiviral software companies. Nevertheless, these findings reveal another aspect of VirusTotal's dynamicity that impacts the verdicts of the scanners it uses: the platform changes the versions of the scanners it uses to scan apps submitted by users. The platform uses the same versions of scanners to label malicious apps targeting different operating systems [116]. Some antiviral scanners may be oblivious to such a change because, for instance, their different versions share the same signatures database. Others, however, may be negatively affected because their versions use different databases, each of which contains signatures of malware targeting a certain platform or operating system. So, this aspect of dynamicity defines the third limitation of the VirusTotal:

VirusTotal Limitation 3

VirusTotal may replace the versions of scanners with inadequate ones that are not designed to detect Android malware based on the request of the scanner's vendor or managing firm, as stated by VirusTotal itself.

5.3. Stability of VirusTotal Scanners

The results in Section 4.3 and Section 5.2 reveal two aspects of VirusTotal's dynamicity that might impact the labeling accuracies of some scanners. In particular, these two aspects cause the accuracies of some VirusTotal scanners to fluctuate. To train reliable ML-based labeling strategies, Maat attempts to rely on VirusTotal scanners that do not often change the labels they assign to Android apps (i.e., stable scanners). Maat identifies stable VirusTotal scanners over a period of time by considering those scanners whose *certainty* score exceeds 0.90. That is *certainty* is a property of one scanner.

A scanner's *certainty* score is calculated as follows. The labels given by the scanner to each app in the training dataset (i.e., *AMD+GPlay*), over a period of time are retrieved. Regardless of the correctness of the labels, the certainty score is calculated by dividing the total number of the **most common** label (i.e., malicious versus benign) by the total number of labels.

As an example, say that the scanner (σ) had the labels $L = \{$malicious, benign, malicious, malicious$\}$ for an app (α) over four points in time. Since the label malicious is the most common label in (L), the certainty score will be three (i.e., counts of malicious), divided by a total of four labels yielding a percentage of 0.75. We can represent this as the

following formula $certainty(\sigma, \alpha) = \frac{count(common(L))}{size(L)}$. To calculate the certainty score for an entire dataset, Maat averages the scanner's certainty scores achieved on individual apps in this dataset.

Similar to the case with the correctness scores, Maat's default threshold of the certainty score is 0.90 to allow for marginal fluctuations in the labels given to apps by scanners, which might be a result of `VirusTotal` excluding those scanners from the apps' scan reports or changing their versions to inadequate ones. So, we define a stable `VirusTotal` scanner in this thesis as follows:

Definition

A stable `VirusTotal` scanner is one that achieves an average certainty score of at least 0.90 on apps in a given dataset over a period of time. This score indicates that, on average and regardless of the correctness of the assigned label, the scanner maintained the same label it assigns to an app 90% of the time.

Using this threshold, Maat retrieved a set of **44 scanners** (i.e., 74.5% of all `VirusTotal` scanners) that had certainty scores of at least 0.90 on apps in the *AMD+GPlay* dataset between November 2018 and July 5^{th}, 2019. One can notice that the number of correct scanners calculated during the same period on the same dataset significantly differs (i.e., 18 versus 44). This is due to correctness and stability adopting different perspectives. On the one hand, correctness is concerned with finding `VirusTotal` scanners that give labels to apps that reflect their ground truth. However, in addition to internal changes of labels, `VirusTotal`'s dynamicity might cause the labels given by scanners and, in turn, their correctness to fluctuate. On the other hand, any definition of stability does not guarantee correctness. For example, over the past three years, we tracked the verdicts given by `VirusTotal` scanners to a repackaged, malicious version[3] of the `K9 Mail` open source app [3] that has been developed by one of our students during a practical course (see Section F). Despite being a malicious app of type `Ransom`, the scanners continued to unanimously deem the app as benign since February 8^{th}, 2017, even after analyzing and re-scanning the app. Another example is an app[4] that we repackaged three years ago; the app continued to be labeled as benign by all scanners until only `K7GW` recognized the app's malignancy in July 2019 and labeled it as a `Trojan`. That is, a stable `VirusTotal` scanner may give consistent labels to an app that are incorrect according to its ground truth. By combining these two concepts as we discuss in Section 5.5, we attempt to find `VirusTotal` scanners that **consistently** give the correct labels to apps.

[3] aa0d0f82c0a84b8dfc4ecda89a83f171cf675a9a

[4] 66c16d79db25dc9d602617dae0485fa5ae6e54b2: A calculator app grafted with a logic-based trigger that deletes user contacts only if the result of the performed arithmetic operation is 50.

Table 5.3.: The evolution of the `VirusTotal` scan reports for `5cfda85debe5e9a7341b4eeed01d92807ed29552` between December 3^{rd}, 2018 and November 8^{th}, 2019.

	2018-12-03	2019-01-08	2019-04-12	2019-04-25	2019-05-09	2019-05-23	2019-06-07	2019-06-21	2019-07-05	2019-07-19	2019-08-02	2019-08-16	2019-08-30	2019-09-13	2019-09-27	2019-10-11	2019-10-25	2019-11-08
positives	39	39	31	33	34	30	32	32	31	32	33	25	30	29	29	26	30	29
total	60	58	57	61	61	60	61	62	60	60	61	56	60	59	59	55	61	61
Negatives (*total-positives*)	21	19	26	28	27	30	29	30	29	28	28	31	30	30	30	29	31	32
positives_delta	2	0	-8	2	1	-3	2	0	-1	1	1	-8	5	-1	0	-3	4	-1
Negatives Delta	×	-2	7	2	-1	3	-1	1	-1	-1	0	3	-1	0	0	-1	2	1

5.4. Stability of `VirusTotal` Scan Reports

In the previous section, we devised a score to estimate the stability of the labels given to one or more apps by **one** `VirusTotal` scanner. However, estimating whether one or more scanners give consistent labels to a dataset of apps does not address the concern of whether a researcher should trust that the `VirusTotal` scan report of an app is stable in light of `VirusTotal`'s dynamicity. That is, we differentiate between the stability of a `VirusTotal` scanner in terms of the consistency of labels it gives to an app (α) over time and the stability of the `VirusTotal` scan report of (α). Addressing this concern requires answering the question of how can we deem a `VirusTotal` scan report of an Android app as stable before using it to label the app itself as malicious or benign (i.e., **(RQ1)**)? In this section, we attempt to address this research question. To answer this question, one needs to define what exactly does *stability* mean. In other words, what are the conditions that need to be achieved in order to deem a `VirusTotal` scan report stable?

In the literature, the primary condition used to define the stability of a `VirusTotal` scan report is the *positives* attribute [91, 63], which depicts the number of scanners deeming an app as malicious. If this number did not change after re-analysis, then the app's scan report can be assumed to have stabilized, even if the scanners that deem the app malicious have changed. This difference in *positives* is captured by another attribute in a `VirusTotal` scan report, viz. *positives_delta*. According to this definition of stability, once the value of *positives_delta* switches to zero, the `VirusTotal` scan report can be considered as stable.

With that in mind, consider the data in Table 5.3, which belongs to an app of type `Ransom` that was first seen on `VirusTotal` on May 11^{th}, 2015. According to the *positives_delta==0* criterion, the app appeared to have stabilized on January 8^{th}, 2019 (i.e., four years after being first seen on `VirusTotal`). However, reanalyzing the app after three months (i.e., on April 12^{th}) indicates that eight scanners seized to deem the app as malicious, rendering the value of *positives_delta* to -8. The number of `VirusTotal` scanners deeming the app as malicious

(*positives*) and the total number of scanners that scanned the app (*total*) continue to fluctuate across different re-analysis points that are merely two weeks apart. This example suggests that a definition of stability according to the criterion *positives_delta==0* is inadequate, given the dynamicity of `VirusTotal`. Similarly, considering the complement of the *positives* attribute (i.e., the number of scanners deeming an app as benign), does not offer a more stable alternative. As seen in the table, the negatives value, calculated as the difference between *total* and *positives*, fluctuates over time, albeit with a relatively smaller delta. In general, the alternative definition of stability should refrain from defining stability in terms of a specific value or attribute (e.g., *positives_delta==0* or *total==61*).

A more relaxed definition of stability of a `VirusTotal` scan report relies on a range of values for one particular attribute, say *positives* rather than an absolute one. That is, instead of deeming a `VirusTotal` scan report as stable once the *positives_delta* hits zero, researchers can define a range of values within which changes in the value of an attribute are tolerated. For example, in Table 5.3, if we tolerate changes in the *positives* attribute within a range of $\pm$10 scanners, then the scan reports of the app `5cfda85debe5e9a7341b4eeed01d92807ed29552` are already stable. Similarly, a range of $\pm$7 scanners for the negative attribute implies that the app's scan reports are already stable. The main problem with such a definition of stability is that it introduces subjectivity. The dynamicity of `VirusTotal` and the possibility of having different ranges per malware types and app ages prevent us from deeming a particular range as a universally adequate one to estimate the stability of `VirusTotal` scan reports.

Assuming that the attribute and the range of its values were standardized (e.g., *positives*$\pm$10 scanners), we still need to examine the `VirusTotal` scan reports of an app over a period of time to ensure that the value of the standard attribute indeed remained within the designated range. For example, how long does *positives* in a given `VirusTotal` scan report need to remain with a range of $\pm$10 scanners before the scan report is deemed stable? Answering this question introduces two problems. Firstly, researchers need to acquire older versions of the `VirusTotal` scan report they are attempting to deem as stable. Unfortunately, in addition to **not** automatically and regularly re-scanning apps, `VirusTotal` does not grant access to the scan history of apps. So, researchers have to build their history of a `VirusTotal` scan report by re-scanning the app and downloading its new scan report moving forward. While this approach is effective for newly-developed Android (malicious) apps, it does not reflect the evolution of the `VirusTotal` scan reports of older apps whose scan reports are less sensitive to `VirusTotal`'s dynamicity, as discussed in Section 4.3. The second problem is that specifying a period of time for which an attribute needs to remain within a particular range for the `VirusTotal` scan report to be deemed as stable is per se subjective and might differ depending on the researcher's views, app's malignancy, app's malware type, and so forth.

VirusTotal Limitation 4

VirusTotal does not grant access to academic researchers to the history of scan reports of apps previously added and scanned on the platform, even if such apps were added by the academic community itself. In fact, we are not sure whether VirusTotal keeps or discards the current scan reports of apps prior to re-scanning apps

Another method to define stability is to designate a VirusTotal scan report of an app as stable after a particular period of time. Within the context of Windows-based malware, Miller et al. suggested that it takes approximately *one year* for VirusTotal scan reports to stabilize [91], whereas Kantchelian et al. argued in [63] that this period can be as brief as *four weeks* [63]. This period of time is usually estimated in terms of attributes found in the VirusTotal scan reports of apps. For example, one can estimate the period it takes for a scan report to stabilize by calculating the difference between the date on which the attribute *positives_delta* converged to zero and the date on which the app was first seen on VirusTotal (i.e., the *first_seen* attribute). As discussed and demonstrated in the previous section, the dynamicity of VirusTotal causes the values of attributes in a VirusTotal scan report to fluctuate. So, even if an attribute reaches a target value, the platform's dynamicity is likely to cause this value to change in the future. Secondly, the choice of an attribute to estimate the time taken for a scan report to stabilize and its target value is subjective.

To conclude this section, in light of VirusTotal's dynamicity and limitations, in order to estimate whether the VirusTotal scan report of an app is stable or not, a researcher has to choose an attribute in the VirusTotal scan report (e.g., *positives*), a range of values within which the value of the chosen attribute needs to remain, and a time period within which the values of the chosen attribute are to be monitored. Given the lack of standards on how to use VirusTotal in general, choosing these values will be delegated to each researcher leading to the same problem with choosing a threshold to label apps as malicious or benign (see Chapter 4). We argue that VirusTotal's dynamicity renders the pursuit of criteria to deem a VirusTotal scan report as stable rather infeasible, which has also been recently confirmed by researchers within the research community [162]. In this context, in designing Maat we differentiate between the stability of an individual VirusTotal scanner and that of a VirusTotal scan report. Maat does not attempt to judge the stability of a VirusTotal scan report prior to using it. Instead, it identifies the scanners that give stable verdicts to apps in the training dataset based on the certainty score defined in Section 5.3. That is, Maat uses the verdicts of stable scanners regardless of whether a scan report has stabilized.

5.5. Features Extracted from Scan Reports

In this section, we recap on the type of features that Maat extracts from the VirusTotal scan reports of its training dataset and relate them to the insights we gained from the measurements performed in the previous sections. As mentioned in Section 5.1, there are two types of features that Maat can extract from scan reports, namely engineered and naive.

5.5.1. Engineered Features

The engineered features Maat extracts from scan reports (listed in appendix G and online) can be divided into three categories. In the first category of features, Maat considers the verdicts of VirusTotal scanners that had a correctness score and a certainty score of at least 0.90. As seen in Section 5.2 and Section 5.3, the performance of some correct scanners, such as F-Secure, fluctuates over time (i.e., not stable), whereas the stability of scanners in general does not guarantee correctness. To mitigate both issues, Maat relies on the verdicts given by the intersection of correct and stable scanners. Taking the intersection between the 18 correct scanners the 44 stable scanners yields a set of 16 VirusTotal scanners that include all the correct scanners apart from F-Secure, which already exhibited fluctuation in detection performance (see Section 5.2), and Trustlook. The reason behind this is that the continuity of correctly labeling apps leads to the stability of labels. In other words, if a scanner is consistently accurate at giving the same, correct label to an app, it is ipso facto a stable scanner. This relationship between correctness and stability, as discussed before, does not go the other way around (i.e., stability does not imply correctness).

The second category of features is based on attributes found in VirusTotal scan reports that imply the age and, perhaps, the maturity of the app and its scan report. Maat extracts the age of the app's scan report as the difference between the extraction date and that of *first_seen* along with the *times_submitted* attribute, the *positives* attribute, and the *total* attribute. In our analysis, we noticed that older malicious apps have higher ranges of *positives* and *total* values than newer ones. For example, as of November 8^{th}, 2019, the malicious apps in the *AMD* dataset have an average *positives* value of 26.26 ± 5.53: a number newly-developed malicious apps and benign ones are unlikely to reach. So, we thought that such values might assist Maat to discern malicious, benign, and ambiguous malicious apps.

The third and last category of engineered features is meant to approximate the structure and behavior of apps. In Chapter 2, we discussed that malicious apps tend to adopt similar structures, re-use libraries, exploit similar vulnerabilities, or share the same codebases. We assume that such trends can be reflected in the permissions these apps request and the tags assigned to them by VirusTotal. So, as part of the engineered features, we include the list of permissions (not) requested by the app and the tags (not) assigned to them by VirusTotal.

In total, Maat extracts 372 features from the VirusTotal scan report of each app. Having discussed the impact of the curse of dimensionality on the performance of ML algorithms,

we implemented Maat to select the most informative features from this feature set. In Chapter 6, we discuss the types of features selected from the entire corpus of engineered features.

5.5.2. Naive Features

Naive features comprise the verdicts of all VirusTotal scanners, regardless of their correctness or stability. With this set of features, as mentioned earlier, we allow Maat's random forests to identify and choose the VirusTotal scanners that train the most effective ML-based labeling strategies. We use this type of feature for three reasons. First, naive features are fast and easy to extract from VirusTotal scan reports. Second, we wish to investigate whether allowing Maat's random forests to select scanners would yield a set of scanners that overlap with the ones identified using the measurements in Section 5.2 and Section 5.3. Third, we wish to assess whether the second and third categories of engineered features help Maat train better ML-based labeling strategies or hinder their performance.

The dimensionality of naive features is 60 scanners. However, we also allow Maat to select the most informative features (i.e., scanner verdicts, in this case). As of November 8^{th}, 2019, using the scan reports of apps in the *AMD+GPlay* dataset, the set of scanners selected from the full corpus of naive features comprised 16 scanners listed in appendix D.

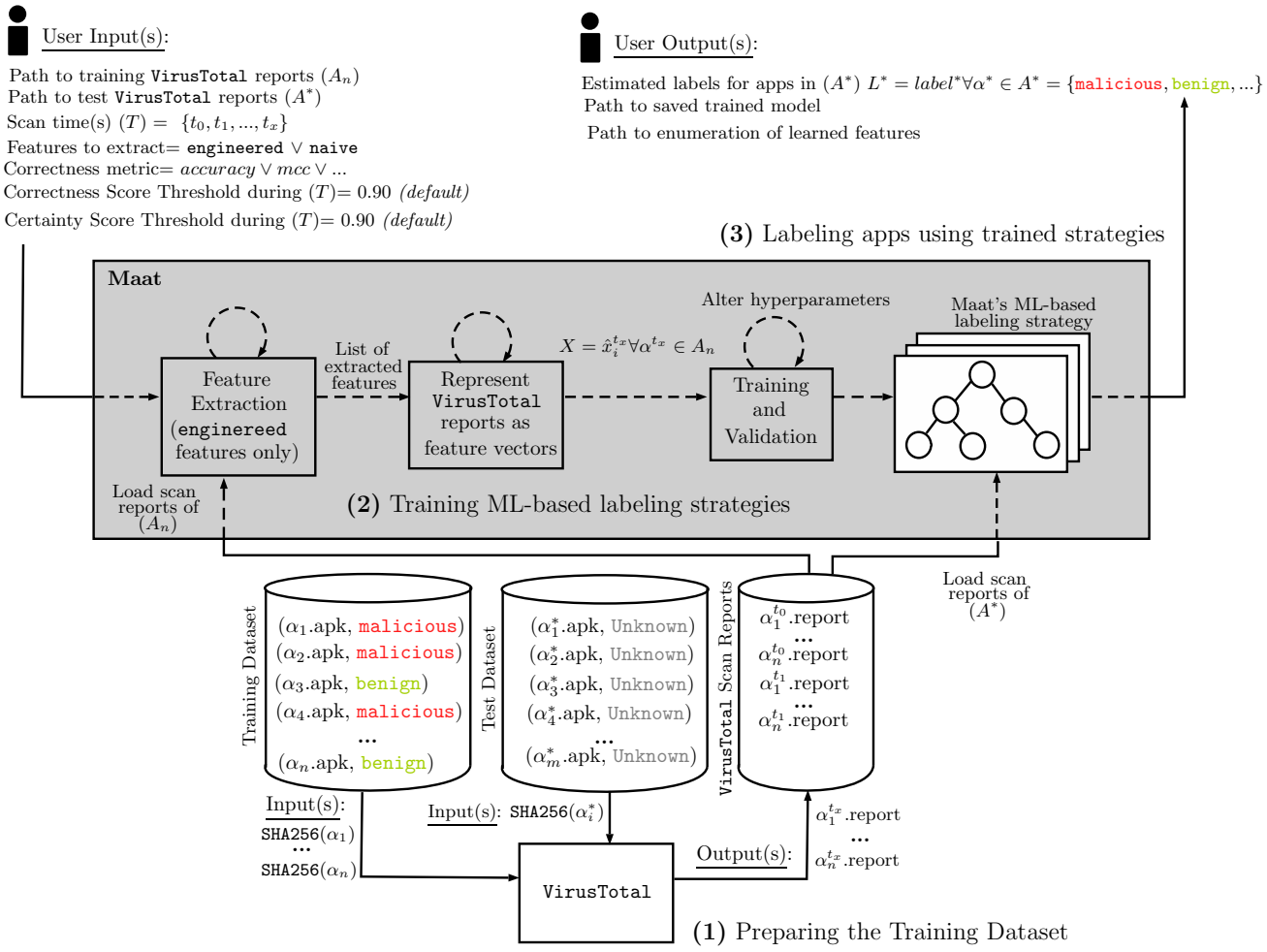
5.6. Using Maat

As discussed in Section 1.3.2, we implemented Maat as an API to support researchers with the analysis of VirusTotal scan reports and the extraction and visualization of insights about VirusTotal scanners over time. For example, in the previous sections, we showed how Maat could be used to find the set of correct and stable VirusTotal scanners based on the VirusTotal scan reports of Android apps in a given dataset over a period of time. In this section, we describe how Maat can be used to train ML-based labeling strategies that are used to label arbitrary Android apps as malicious and benign based on their VirusTotal scan reports. We do not describe the technical details of what commands to issue or how to implement source code that leverages Maat's API to achieve the aforementioned tasks; these details can be found on Maat's online (https://github.com/tum-i22/Maat) user manual. In Figure 5.3, we visualize the typical process of using Maat to train ML-based labeling strategies to label apps based on their VirusTotal scan reports. We detail this process in the following sections.

5.6.1. Preparing the Training Dataset

Maat's users need to gather the scan reports of pre-labeled Android malicious and benign apps that Maat can use as the training dataset for ML-based labeling strategies. Maat does **not** extract features from the APK archives of apps. So, the APK archives of those apps

Figure 5.3.: Maat's process of training ML-based labeling strategies to label Android apps based on their `VirusTotal` scan reports.



do not need to be downloaded; any unique representation of the apps (e.g., Secure Hash Algorithms (SHA) or Message Digest (MD5) hashes of their content), would suffice. Given the APK archives or hashes of apps in the training dataset, Maat's users need to possess at least one version of the VirusTotal scan reports of those apps (i.e., at time t_0). This can be acquired by downloading the current version of VirusTotal scan reports of the apps in the dataset. The downloaded reports are used to extract the features discussed in the previous section. At the end of the feature extraction phase, the training dataset used by Maat to train ML-based labeling strategies would comprise the following. For each app (α) in the dataset (A), using **at least one** VirusTotal scan report of the app ($\alpha^{t_0}.report$), Maat extracts features from the report and yields a vector of numerical feature ($\hat{x}_{\alpha^{t_0}}$) and a label (y_α) that depicts the class of (α) (i.e., malicious or benign). The accuracy of the labels assigned to apps and their feature vectors in the training dataset is the responsibility of Maat's users. Users can either manually analyze the apps themselves or rely on previous (manual) analysis. In this thesis, for example, we rely on the manual analysis conducted by Wei et al. in [153] to label apps in the *AMD* dataset as malicious.

Maat can use just one scan report per app to train its ML-based labeling strategies. However, the framework can also use multiple versions of VirusTotal scan reports per app that depict the labels given by VirusTotal scanners to each app at different scan dates. Using multiple scan reports per app is used to find VirusTotal scanners that are correct and stable over time, which makes the results more resilient to VirusTotal's fluctuations. This is relevant to users wishing to train ML-based labeling strategies using engineered features. However, Maat users wishing to use multiple scan reports per app to train ML-based labeling strategies can only gather future scan reports because VirusTotal does not grant access to the history of scan reports per app. In this thesis, we gathered the VirusTotal scan reports of apps in the *AMD+GPlay* dataset starting from late November 2018 to July 5^{th}, 2019 to be able to compare the performance of ML-based labeling strategies trained using engineered and naive features.

5.6.2. Training ML-based Labeling Strategies

Once the training dataset is ready and prepared, the users of Maat can use its API to train ML-based labeling strategies. The only other input that Maat users need to specify is the type of features they wish to be extracted from the VirusTotal scan reports in the training dataset. If the user opts to utilize the engineered features, then they can specify the correctness metric to be used (e.g., accuracy versus MCC), the threshold of correctness needed to be met by a VirusTotal scanner to be considered as a feature (defaults to 0.90), and the threshold of the certainty score needed to be met by a VirusTotal scanner to be considered as a feature (also defaults to 0.90). No parameter values need to be specified in the case of utilizing naive features.

As mentioned in Section 5.1, Maat handles the processes of selecting the more informative features extracted from the scan reports and the hyperparameters of the ML model it returns, which is a random forest of a 100 decision trees (by default). At the end of the training

phase, Maat returns a random forest that achieved the best training accuracy on the training dataset after using 10-Fold cross-validation. The random forest is serialized and stored on the user's disk, along with the list of features that are used by this random forest. Effectively, Maat relieved the user of the burden of having to decide upon the attributes or features to extract from the VirusTotal scan reports of training apps. Even in the case of engineered features, which are manually engineered, Maat selects a subset of those features that are more informative than others without user intervention. Furthermore, we implemented Maat to yield the same random forests upon using the same training dataset and labels to enable the reproduction of results by different researchers using the same training dataset and labels.

We consider Maat and its ML-based labeling strategies to be a complement the brute-forcing algorithm introduced in Section 4.4 to find the currently optimal threshold of VirusTotal scanners to label apps instead of choosing a subjective fixed threshold in terms of standardizing the process of interpreting VirusTotal's scan reports to label Android apps **(RQ4)**. However, Maat's ML-based labeling strategies are built to be more resilient to the dynamicity of VirusTotal than their threshold-based counterparts. This is made possible by relying on the verdicts of 100 decision trees that comprise the verdicts of multiple VirusTotal scanners that Maat found reliable during the training phase. Consequently, VirusTotal's manipulation of the versions or verdicts of scanners will be confined to a subset of the aforementioned decision trees, which we hypothesize that it does not have a noticeable impact on the random forest's labeling accuracy. We test this hypothesis in Chapter 6. Thus, using Maat's ML-based labeling strategies, there is no need to regularly acquire the current VirusTotal scan reports of apps in the training dataset and retrain the random forests. In other words, we hypothesize that users can use Maat's ML-based labeling strategies trained at one point in time to label Android apps for longer periods than the optimal thresholds acquired via the brute force algorithm. As for the diversity of the training dataset used by Maat, the results discussed in Section 4.3 and tabulated in Table 5.1 suggest that there is a subset of VirusTotal scanners that prove to be accurate at labeling apps regardless of the age and composition of the dataset. We implemented Maat to attempt to identify those scanners and rely on their verdicts in the random forests it trains. So, we hypothesize that the age of apps used in Maat's training dataset and the maturity of their VirusTotal scan report may not impact their labeling performance against new apps (e.g., in the *Hand-Labeled 2019* dataset). If true, then unlike the brute force algorithm, Maat's training dataset need not be updated and labeled on regular basis to be able to accurately label newly-developed Android apps based on their VirusTotal scan report.

5.6.3. Labeling Apps Using Maat's ML-based Labeling Strategies

Given the path to the file containing the serialized ML-based labeling strategy, the path to the features this labeling strategy uses, and the path to the VirusTotal scan report of a test app, Maat will attempt to label the app as malicious or benign as follows. First,

Maat loads the ML-based labeling strategy and deserializes it into a random forest classifier. Second, Maat parses the features stored into the features file. Third, the framework extracts the features expected by the loaded random forest from the test app's `VirusTotal` scan report and represents them as a feature of numerical features. Lastly, Maat uses the random forest to predict the class (i.e., malicious or benign), of the feature vector and returns the result to the user.

This process is used by Maat as part of two experiments that the framework currently supports by Maat. On the one hand, users can use a pre-trained Maat's ML-based labeling strategy to label a dataset of test apps and compare the predicted labels to some ground truth. In this case, the user is assessing the labeling accuracy of Maat's labeling strategies. On the other hand, users can use pre-trained labeling strategies to label apps whose ground truth is unknown. Maat uses the predicted labels alongside features extracted from those apps (which are to be supplied by the user) to train and validate ML-based detection methods. This type of experiment emulates the process of a researcher labeling a newly-acquired dataset of Android apps to evaluate a ML-detection method they implemented. The evaluation of the trained ML-based detection method can be carried out against test apps whose ground truth labels are known to assess the impact of Maat's labeling strategies on the detection ability of the ML-based detection methods.

For convenience, we implemented tools that utilize Maat's API to automate both experiments. That is, the tools handle the training of Maat's ML-based labeling strategies using the specified training dataset, labeling (test) apps, training ML-based detection methods based on those labels and supplied feature vectors, labeling test apps using the trained detection methods, and visualizing the results. As mentioned earlier in this section, Maat's user manual, documentation, and source code can be found on Maat's website (https://github.com/tum-i22/Maat).

5.7. Summary

One main objective of this thesis is to standardize the utilization of `VirusTotal` scan reports to label (Android) apps in ways that bypass `VirusTotal`'s dynamicity. In Chapter 4, we identified aspects of the platform dynamicity that negatively impact threshold-based labeling strategies and proposed an algorithm to find the currently optimal thresholds that should be used by researchers to label apps based on their `VirusTotal` scan reports. However, to cope with the dynamicity of `VirusTotal`, the proposed brute-force-based algorithm suffered from its own limitations, including the continuous acquisition and (manual) labeling of new apps used by the algorithm to find the currently optimal threshold. To complement the brute force algorithm and mitigate its limitations, we implemented Maat. Via its API and tools that utilize it, Maat is designed to help the research community address some of the issues we discussed in Section 1.3.1, such as finding indications of whether a `VirusTotal` scan report has stabilized **(RQ1)**, whether there exists a set of `VirusTotal` scanners whose verdicts are consistent and correct across different datasets

(RQ3), and further unveiling aspects of VirusTotal's dynamicity that might hinder accurately labeling Android apps as malicious and benign. More importantly, Maat enables researchers to train ML-based labeling strategies that focus on the verdicts of correct and stable VirusTotal scanners rather than a number of random scanners deeming an app as malicious. We hypothesize that the structure of Maat's ML-based labeling strategies make them less susceptible to VirusTotal's dynamicity and can, hence, be used to accurately label apps based on their VirusTotal scan reports for longer periods of time without having to constantly re-scan apps, download their up-to-date VirusTotal scan reports, and retraining the labeling strategies. Effectively, this should make Maat's ML-based labeling strategies a viable alternative to their threshold-based counterparts, including ones that are devised using our proposed brute force algorithm. In the next chapter, we evaluate Maat's ML-based labeling strategies to test the aforementioned hypotheses.

6. Evaluating Maat

This chapter evaluates Maat*'s ability to assign labels to apps based on their* `VirusTotal` *scan reports that accurately reflect their ground truths. The chapter also discusses the impact of such accurate labeling on the performance of ML-based detection methods and, in general, the role of accurate labeling on the performance of detection methods. Parts of this chapter have previously appeared in peer-reviewed publications [9] and [116], co-authored by the author of this thesis.*

Given the popularity of `VirusTotal` and the unlikelihood of it being replaced anytime soon, in Section 1.3, we set the main objective of this thesis to find or devise methods that help the research community with optimally utilizing `VirusTotal` given its dynamicity and limitations. One important aspect of such an optimal utilization is to devise methods that standardize the interpretation of `VirusTotal` scan reports to label (Android) apps as malicious or benign and yield labels that better reflect the malignancy of those apps (i.e., **(RQ4)**). To that end, in Chapter 4, we proposed an algorithm that can identify the currently optimal number of `VirusTotal` scanners that can be used by threshold-based labeling strategies to label apps as malicious and benign. Unfortunately, due to `VirusTotal`'s dynamicity and limitations, the proposed algorithm can only be useful under specific circumstances that cannot be met by all researchers viz., possession of diverse datasets of Android apps and their `VirusTotal` scan reports, access to commercial licenses, and significant manpower to manually label apps. In Chapter 5, we described a framework, Maat, that we implemented to automate the process of analyzing `VirusTotal` scan reports and train ML-based labeling strategies with minimal intervention from its users. We hypothesize that Maat and the ML-based labeling strategies it trains can mitigate the limitations of the proposed algorithm and provide the research community with a standard, effective method to interpret `VirusTotal` scan reports to label (Android) apps accurately. This makes our evaluation of Maat towards verifying this hypothesis take **two dimensions**. On the one hand, we attempt to verify whether Maat indeed mitigates the limitations of the brute force algorithm proposed in Section 4.4. On the other hand, we wish to evaluate Maat ML-based labeling strategies' performance in comparison to conventional threshold-based labeling strategies so that we could declare it as a viable replacement for the subjective labeling strategies. To give a clear structure to our evaluation, in this chapter, we extend the research questions **(RQ4)** and **(RQ5)** and address each one in a separate section.

Concerning the first dimension of evaluating Maat's ML-based labeling strategies, recall that the main limitations of the brute force algorithm proposed in Section 4.4 are two-fold.

Firstly, the algorithm required the existence of a diverse dataset of pre-labeled malicious and benign apps that should always be extended with newly-acquired and labeled apps. Secondly, to be able to calculate the currently optimal threshold of VirusTotal scanners, the apps in the aforementioned dataset need to be re-scanned via VirusTotal and their scan reports need to be downloaded. For Maat's ML-based labeling strategies to mitigate those limitations, it should (a) not require the frequent acquisition of new apps to be included in its training dataset, and (b) be able to accurately label apps for longer periods of time before needing re-training. We capture these two requirements via the following research questions:

RQ4.1: For how long can a Maat trained ML-based labeling strategy maintain the correct labels it assigns to apps?

RQ5.1: How does VirusTotal's dynamicity impact the structure and performance of ML-based labeling strategies?

Mitigating the limitations of the brute force algorithm does not necessarily imply that Maat's ML-based labeling strategies are a viable alternative to threshold-based labeling strategies and have the potential of being the standard method used to label (Android) apps based on their VirusTotal scan reports. These ML-based labeling strategies should also prove to be better than their conventional threshold-based counterparts in terms of assigning labels to apps that better reflect their ground truth. The second dimension of evaluating Maat's ML-based labeling strategies is concerned with testing whether these strategies can outperform threshold-based labeling strategies. In particular, we assess the performance of Maat's ML-based labeling strategies against that of threshold-based labeling strategies, including ones adopting the currently optimal threshold according to the brute force algorithm according to two criteria. In Section 6.1, we verify whether Maat's ML-based labeling strategies trained at one point in time can maintain labeling accuracies in the future that rival those achieved using threshold-based labeling strategies that adopt the current optimal thresholds. The second set of experiments in Section 6.4 compares the impact of threshold-based labeling strategies versus that of Maat's ML-based labeling strategies on the effectiveness of ML-based detection methods against out-of-sample apps over time. With these two sets of experiments, we wish to answer the following research questions:

RQ4.2: How does the labeling accuracy of Maat's ML-based labeling strategies compare to the accuracy of threshold-based labeling strategies over a period of time?

RQ4.3: What kind of features are learned by Maat's ML-based labeling strategies? And how do they impact the performance of those strategies?

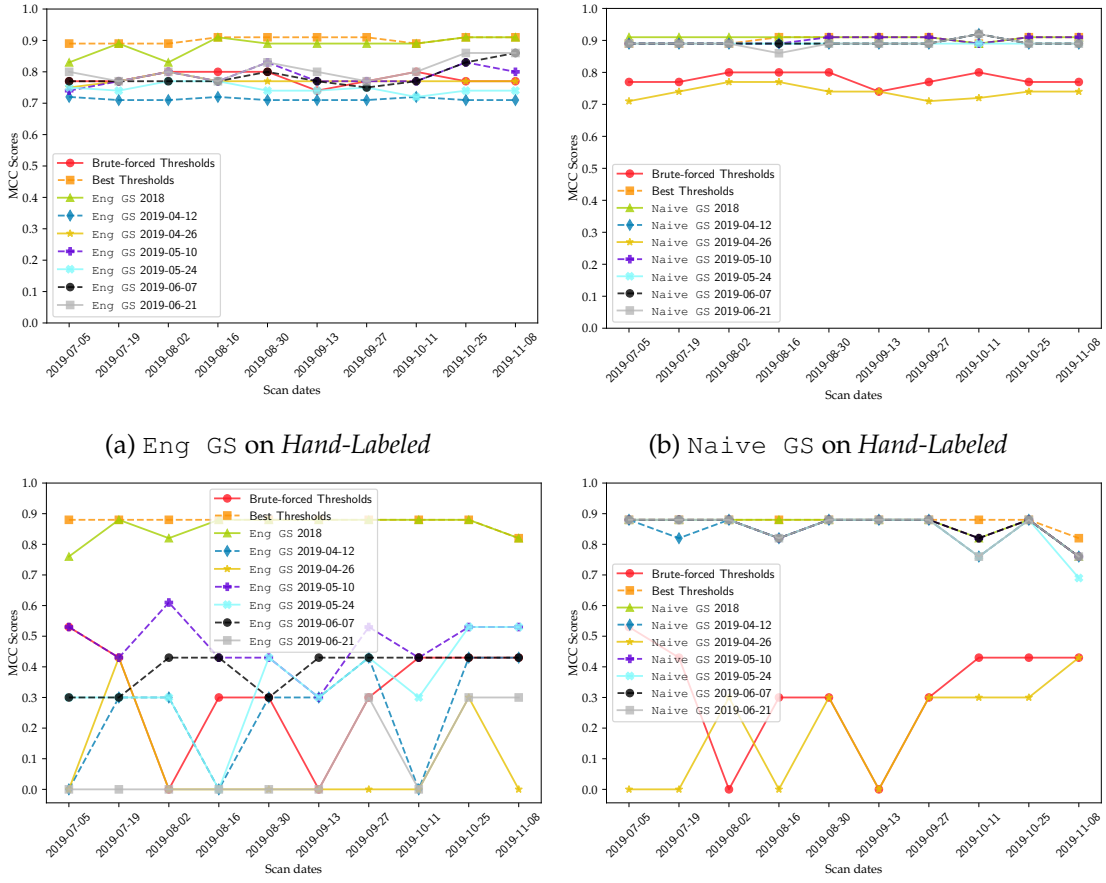
RQ4.4: What is the impact of Maat's ML-based labeling strategies on the performance of ML-based detection method in comparison to that of threshold-based labeling strategies?

6.1. Accurately Labeling Apps

In this set of experiments, we attempt to address the research questions **(RQ4.1)** and **(RQ4.2)**, namely whether Maat's ML-based labeling strategies trained at one point in time can sustain a labeling accuracy similar to or better than threshold-based labeling strategies for longer periods of time. We trained Maat ML-based labeling strategies using `VirusTotal` scan reports of apps in the *AMD+GPlay* dataset downloaded on November 2018, and used the trained strategies to label apps in the test datasets *Hand-Labeled* and *Hand-Labeled 2019* datasets between July 5^{th}, 2019 and November 8^{th}, 2019. In the experiments, we used ML-based labeling strategies that use the two types of features mentioned in Section 5.5 (i.e., engineered versus naive) and trained using the technique of grid search. The reason behind focusing on grid search is that–although it takes between three and seven hours to find the best random forests–it guarantees to find the best solution, whereas randomized search favors pace and returns the best random solution it found within a given period of time or number of attempts. That is, every time a randomized search technique is executed, a different random forest is returned. We also included labeling strategies that used the `SelectFromModel` technique to select the most informative features. For readability, we shorten the names of ML labeling strategies in the following manner: engineered and naive features are referred to as `Eng` and `Naive`, respectively, grid search and randomized search are referred to as `GS` and `RS`, respectively, and strategies that use selected features are referred to using `Sel`. For example, a labeling strategy that uses selected engineered features and the grid search techniques will be referred to as `Eng Sel GS`.

In addition to the ML-based labeling strategies, we calculate the labeling accuracies of two types of threshold-based labeling strategies against apps in the *Hand-Labeled* and *Hand-Labeled 2019* datasets. First, we use a threshold-based strategy that uses the optimal threshold at each scan date (between July 5^{th}, 2019 and November 8^{th}, 2019) identified using the brute force algorithm (see Algorithm 1) on the *AMD+GPlay* dataset. We refer to this labeling strategy as the *Brute-forced Thresholds*. Second, we use labeling strategies that use the thresholds that would achieve the best labeling accuracy (in terms of the MCC score) on both of the *Hand-Labeled* and *Hand-Labeled 2019* datasets. These labeling strategies, which we refer to as *Best Thresholds*, are meant to simulate a scenario in which a researcher always manages to find the thresholds that yield the best possible MCC scores on the test datasets. To the best of our knowledge, there are no methods discussed in the literature that can do that. Thus, the performance of these thresholds depicts an upper bound on the performance of threshold-based labeling strategies on the test datasets. By comparing the labeling performance of Maat's ML-based labeling strategies to that of the Best Thresholds, we are effectively comparing the performance of the former to the best possible performance of a threshold-based labeling strategy, as part of assessing the potential of the ML-based labeling strategies to replace threshold-based labeling strategies.

In Figure 6.1 and Figure 6.2, we plot the labeling accuracies of ML-based, Brute-forced, and Best Threshold labeling strategies in terms of the MCC score against apps in the *Hand-Labeled* and *Hand-Labeled 2019* datasets between July 5^{th}, 2019 and November 8^{th},



(a) `Eng GS` on *Hand-Labeled*

(b) `Naive GS` on *Hand-Labeled*

(c) `Eng GS` on *Hand-Labeled 2019*

(d) `Naive GS` on *Hand-Labeled 2019*

Figure 6.1.: The labeling accuracies achieved by Maat's ML-based labeling strategies `Eng GS` and `Naive GS` trained with scan reports downloaded in between November 2018 and June 21st, 2019 against apps in the *Hand-Labeled* and *Hand-Labeled 2019* datasets over time.

2019. Each of the ML-based labeling strategies used in this experiment were trained using VirusTotal scan reports of apps in the *AMD+GPlay* dataset **at one point in time** between April 12^{th}, 2019 and June 21^{st}, 2019. For example, the green line with triangles depicts the labeling performance of Maat ML-based labeling strategies trained using VirusTotal scan reports of apps in the *AMD+GPlay* dataset downloaded in November 2018. The difference between the two figures is that Figure 6.2 plots the labeling accuracy of ML-based labeling strategies trained using selected features extracted from the VirusTotal scan report rather the full corpus of features (see Section 5.5).

We made a number of observations based on the MCC scores in Figure 6.1 and Figure 6.2, that we represent as an enumeration for better readability.

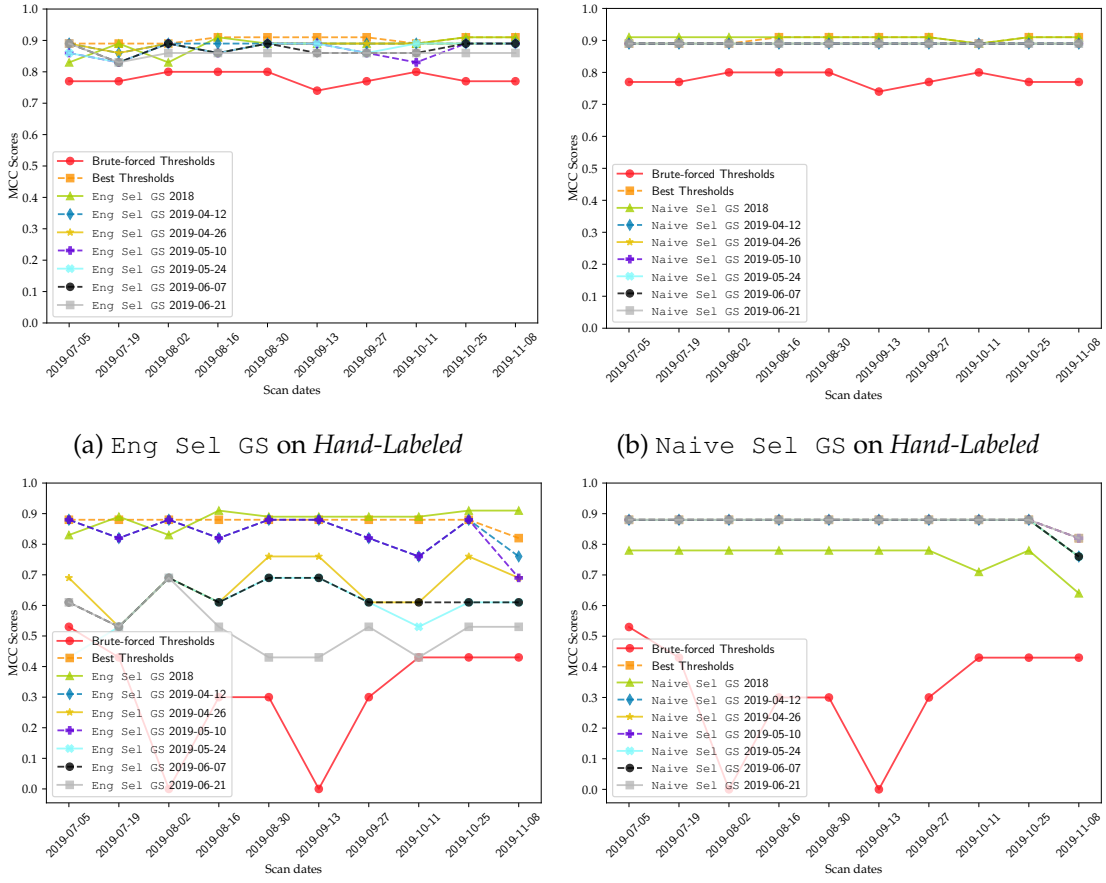
1. The labeling accuracy of Brute-forced labeling strategies is among the worst-performing labeling strategies, especially against apps in the *Hand-Labeled 2019* dataset. This performance demonstrates the need for a temporally diverse dataset in order for the brute force algorithm to be useful, as discussed in Section 4.4. Given that the *AMD+GPlay* dataset does not contain apps developed in 2019, the identified thresholds were high (i.e., between 10 and 13 scanners), which are much higher than the current values of the *positive* attribute in the VirusTotal scan reports of the newly-developed apps in the *Hand-Labeled 2019* dataset.

2. The MCC scores of Maat's ML-based labeling strategies mimic–and often overcome– the scores of the Best Threshold labeling strategies, especially upon using the naive features.

3. In general, ML-based labeling strategies trained using naive features appear to achieve labeling accuracies on both test datasets *Hand-Labeled* and *Hand-Labeled 2019* that are more stable than their counterparts using engineered features.

4. Lastly, the performance of both types of features seems to fluctuate over time, albeit not with the same rate or severity. Using selected features seems to decrease or remove such fluctuations, especially upon using the naive feature set. Furthermore, the fluctuations are usually more visible in the MCC scores resulting from labeling apps in the *Hand-Labeled 2019* dataset. As discussed and demonstrated in Section 4.3, VirusTotal's manipulation of scanners in the scan reports of newly-developed apps has a greater impact on the accuracies of labeling strategies, especially upon labeling malicious apps given that only a small subset of VirusTotal scanners label them malicious. On November 8^{th}, 2019, for example, this aspect of VirusTotal's dynamicity prevents almost all labeling strategies from correctly labeling two out of ten malicious apps in the *Hand-Labeled 2019* datasets after removing three scanners from their scan reports that correctly labeled them as malicious.

With these observations, we can address the research questions **(RQ4.1)** and **(RQ4.2)** as follows. Regarding **(RQ4.1)**, we found that using the naive feature set to train Maat ML-based labeling strategies enables those strategies to maintain a steady labeling accuracy that



(a) `Eng Sel GS` on *Hand-Labeled*

(b) `Naive Sel GS` on *Hand-Labeled*

(c) `Eng Sel GS` on *Hand-Labeled 2019*

(d) `Naive Sel GS` on *Hand-Labeled 2019*

Figure 6.2.: The labeling accuracies achieved by Maat's ML-based labeling strategies `Eng Sel GS` and `Naive Sel GS` trained with scan reports downloaded in between November 2018 and June 21st, 2019 against apps in the *Hand-Labeled* and *Hand-Labeled 2019* datasets over time.

is similar to that of the upper bound of labeling accuracy (i.e., the MCC scores of the Best Threshold labeling strategies). With a few exceptions, the steady performance of ML-based labeling strategies spanned periods between one year (i.e., between November 2018 and November 8^{th}, 2019), and seven months (i.e., between April 12^{th}, 2019 and November 8^{th}, 2019), which answers the research question. These periods, however, are subject to two types of constraints. Given `VirusTotal`'s dynamicity, we cannot guarantee the steadiness of the performance of ML-based labeling strategies in the future. So, we can only treat the period of seven to 12 months as an upper bound. Moreover, we cannot generalize the claim that Maat's ML-based labeling strategies can maintain similar labeling accuracy for a period of seven to 12 months after training. Instead, we should confine this period to the training dataset *AMD+GPlay* and the test datasets *Hand-Labeled* and *Hand-Labeled 2019* datasets. Thus, the answer to **(RQ4.1)** is that–using the aforementioned training and test datasets–Maat ML-based labeling strategies can maintain similar labeling accuracy for a period between seven and 12 months after training, unlike the Brute-forced thresholds that need frequent re-training.

The answer for **(RQ4.2)** is two-fold. Firstly, we noticed that the labeling accuracy of Maat's ML-based labeling strategies in terms of the MCC score is generally better than that of the Brute-force thresholds on both test datasets over time. Secondly, using the naive feature set, the ML-based labeling strategies achieve MCC scores that are similar to the upper bound of scores represented by the performance of the Best Thresholds labeling strategies. In the next section, we attempt to understand the reasons behind the fluctuating performance of ML-based labeling strategies using engineered features as opposed to those using the naive ones.

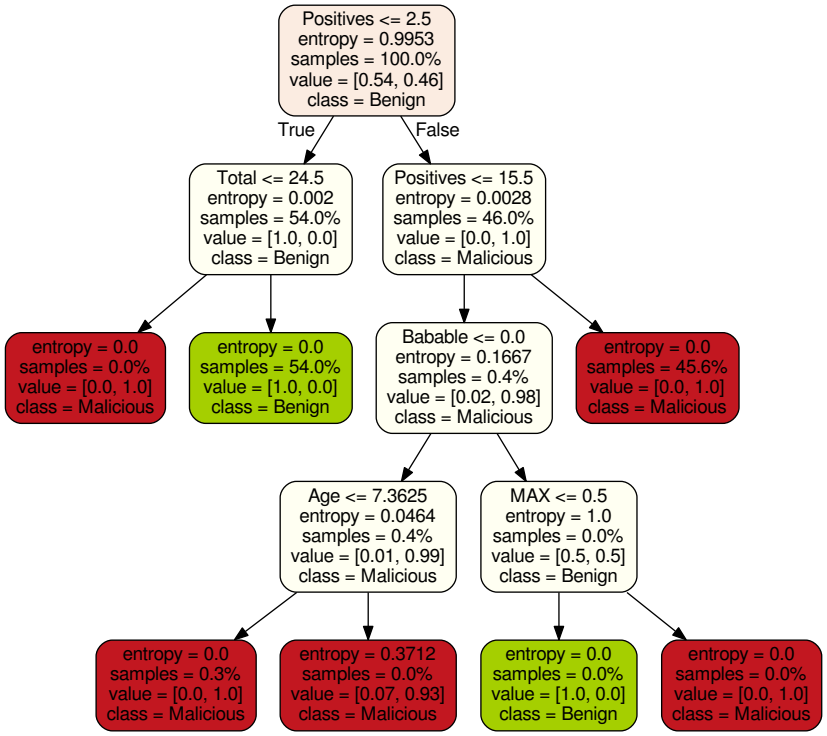
6.2. Features Learned by ML-based Labeling Strategies

In the previous section, we noticed that the labeling accuracy of Maat's ML-based labeling strategies using the naive features set is, by and large, more stable over time and closer to the upper bound MCC scores achieved by the Best Threshold labeling strategies. In this section, we attempt to identify the reasons behind these performances by examining the features learned by the random forests constituting the ML-based labeling strategies, effectively addressing **(RQ4.3)**.

6.2.1. Engineered Features

In Figure 6.3, we show an example of the decision trees trained using engineered features based on `VirusTotal` scan reports of the training dataset *AMD+GPlay* downloaded in November 2018. We found that on this date, the decision trees within the train random forests had almost identical structures and features, whether trained using the full or selected corpus of the engineered feature set (i.e., `Eng GS` or `Eng Sel GS`). As seen in the figure, the tree uses a mixture of the feature categories mentioned in Section 5.5.1, namely

Figure 6.3.: An example of a decision tree trained using grid search and all engineered features `Eng GS` extracted from apps in the *AMD+GPlay* dataset in November 2018.



attributes found in the apps' scan reports (e.g., *positives* and *total*), features that indicate the age of an app, and the verdicts of some `VirusTotal` scan reports (e.g., `Babable` and `MAX`). The tree's structure allows it to accurately classify Android apps in the *Hand-Labeled* and *Hand-Labeled 2019* datasets as follows. First, the left subtree classifies apps after checking the ratio of *positives* to *total* attributes in the scan report, which allows it to cater to newly-developed malicious apps. That is if the number of scanners is less than 2 (i.e., *positives*$\leq$2.5) and the total number of scanners in the scan reports is less than 24 (i.e., *total*$\geq$24.5), the tree assumes that the app is newly-developed yet around 9% of scanners deem it malicious, and in turn, labels the app as malicious. Otherwise, if the total number of scanners is more than 24, the tree considers the app's scan report to be mature (i.e., old app), yet only a small subset of scanners deem it as malicious; in this case, the tree considers the *positives* to be false positives and deems the app as benign. For our test datasets, we found that this subtree is effective at classifying benign apps and minimizing false positives.

The tree's right subtree checks the *positives* attribute again and labels apps as malicious if more than 16 scanners deem an app malicious. This check helps identify old malicious apps whose `VirusTotal` scan reports have matured to include values of *positives* higher than these thresholds. As per November 8^{th}, 2019, 99.65% of the malicious apps in the *AMD+GPlay* dataset and 67% of those in the *Hand-Labeled* test dataset had *positives* values greater than or equal to 16. However, this check does not help classify newly-developed malicious apps, such as those in the *Hand-Labeled 2019* dataset. In this case, the tree checks the verdict of the `Babable` scanner. If the scanner deems an app malicious, it checks the verdict of another scanner (i.e., TrendMicro's `MAX`), and deems the app as malicious or benign according to this scanner's verdict. Effectively, the tree deems the app as malicious if two scanners, which we found to be among the correct scanners for the *AMD+GPlay* and *Hand-Labeled 2019* datasets in Section 5.2, deem the app malicious. If `Babable` finds the app benign, the tree makes a final check about the age of the app being less than 7.3 years. However, this check can be ignored because, regardless of the outcome, the tree assigns the app to the majority class in this subtree, which is malicious.

For apps in our test datasets, we found that benign and malicious apps are labeled according to the following decision paths. On the one hand, benign apps are classified using the left subtree after satisfying the conditions that *positives*≤ 2 *AND total* ≥ 25. On the other hand, malicious apps in the *Hand-Labeled* dataset were labeled malicious after satisfying the conditions *positives* ≥ 3 *AND positives* ≥ 16. However, given their novelty, malicious apps in the *Hand-Labeled 2019* datasets were labeled malicious after satisfying the conditions *positives*≥ 3 *AND* `Babable` ≤ 0.0. Upon further investigation of the app's scan reports, we found that they are correctly classified whenever `VirusTotal` excludes `Babable` from their scan reports. Furthermore, we found that the `MAX` scanner never detected any of those apps. So, effectively, the apps were labeled as malicious if the check `Babable` $== -1$ is true. Although this check is effective at labeling apps as malicious, it does not make sense to label apps based on the absence of a scanner. This is an example of how `VirusTotal`'s first limitation of including and excluding scanners in the scan reports of apps might impact the way `Maat`'s ML-based trees are trained. To conclude, it appears that combining different attributes found in the `VirusTotal` scan reports of apps in the *AMD+GPlay* dataset enables ML-based labeling strategies to maintain a steady and decent labeling accuracy over time.

As for the performance of ML-based labeling strategies trained at different points in time, we found that the main reason behind their underperformance is their shallowness. We found that–whether they use all of the engineered features or a selected subset of them–all decision trees in random forests trained between April 12^{th}, 2019 and June 7^{th}, 2019 employ only one check making them shallow trees of depth[1] one. In Section 6.3, we explain the reasons that led to training random forests with such a shallow depth. With this depth, the

[1]The conventional definition of a *depth* is associated with a node in a tree. That is, the depth of a node is the number of edges from the node to the tree's root node. In this section, we use depth to refer to the **maximum number of checks** employed by a decision tree to make a decision

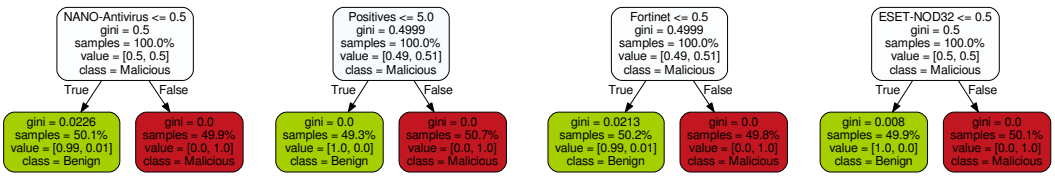
(a) (b) (c) (d)

Figure 6.4.: Four randomly selected decision trees in the ML-based labeling strategies' random forests trained using grid search and **all** engineered features `Eng GS` extracted from `VirusTotal` scan reports of apps in the *AMD+GPlay* dataset downloaded on April 12^{th}, 2019.

tree is unable to represent apps in the training dataset, which is apparent in the impurity of the leaf nodes. For example, the left leaf node in Figure 6.4b shows that 54% of the remaining apps are benign, and 46% are malicious; apps will be labeled as benign because the majority class, in this case, is benign. Later in this chapter, we explain why trees of depth one can be trained using either type of features, viz. engineered and naive.

Despite being all of the depth one, some of the decision trees in these random forests are more effective than others. For example, the decision tree in Figure 6.4a uses the verdict of `ESET-NOD32` to classify apps, which we showed in Section 4.2 and Section 5.2 to be a consistently correct scanner. While effective, this type of tree is susceptible to `VirusTotal`'s first limitation: frequent change in the set of scanners included in scan reports. So, for it to be effective, this tree needs to exist with similar trees that use the verdicts of other reliable scanners. Unfortunately, as seen in Figure 6.4b and Figure 6.4c, other decision trees trained on April 12^{th}, 2019 use features that fail on their own to represent patterns found in either the training dataset *AMD+GPlay* or the test datasets. For example, in Figure 6.4c, the decision tree relies on the *checks-gps* tag to label apps. This `VirusTotal` tag does not help discern the malignancy of an app, especially since using the GPS module is common among malicious and benign apps alike. In fact, relying on such tags might lead to noisy trees, such as the one in Figure 6.4d, which further decreases the overall performance of the random forest.

As seen in Figure 6.5, using a selected subset of the engineered features yields decision trees that are confident about their labels, as indicated by the Gini values in the leaf nodes. Checks that did not yield confident labels, such as the `VirusTotal` tags and the permissions requested by an app, are excluded. The decision trees rely on either the verdicts of scanners that we found to be correct in Section 5.2 on the *AMD+GPlay* dataset or scan report attributes such as *positives* and *total*. While selecting features helped increase the performance of ML-based labeling strategies using engineered features, they still failed to perform well on the *Hand-Labeled 2019* dataset. The reason for this is two-fold. Firstly, relying



(a) (b) (c) (d)

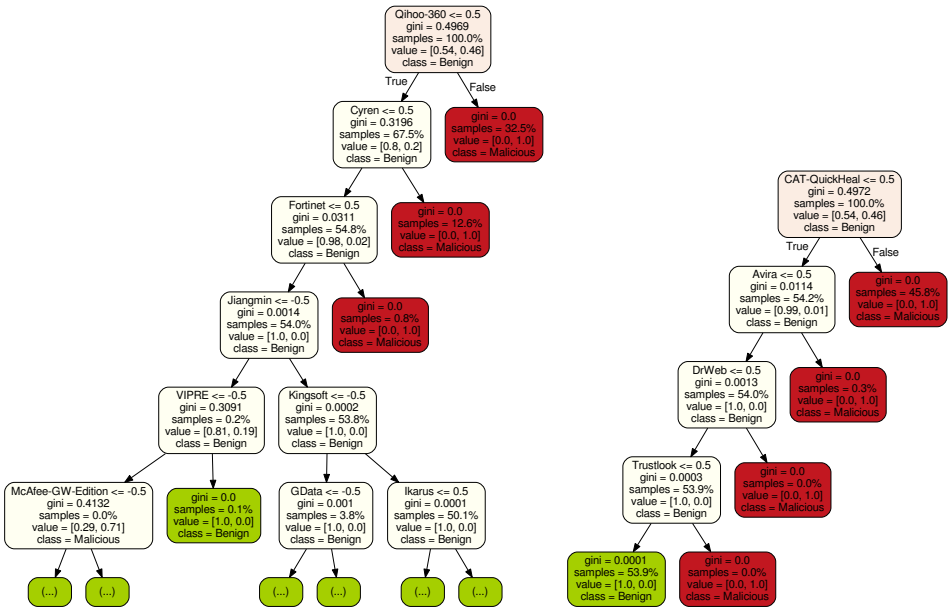
Figure 6.5.: Four randomly selected decision trees in the ML-based labeling strategies' random forests trained using grid search and **selected** engineered features `Eng Sel GS` extracted from `VirusTotal` scan reports of apps in the *AMD+GPlay* dataset downloaded on April 12^{th}, 2019.

on the verdicts of only one scanner makes the decision tree susceptible to `VirusTotal`'s dynamicity and the resulting first limitation. Regardless of the correctness and stability of the scanner, it risks being excluded from the scan reports of apps, which forces a decision tree to label all apps as benign. Secondly, the scan report attributes of *positives* and *total* are also susceptible to `VirusTotal`'s dynamicity. More importantly, using threshold values such as five (i.e., Figure 6.5b) might suit the older apps in the *AMD+GPlay* dataset, where the malicious apps have *positives* values usually greater than ten and benign values of zero *positives*. However, such low values may not generalize to newer apps where only a small subset of `VirusTotal` scanners might already recognize their malignancies.

To conclude this section, we found that Maat ML-based labeling strategies using engineered features can yield good labeling accuracies on our test datasets that are steady over time if the depth of these trained decision trees is more than one. This depth enables the random forests constituting the ML-based labeling strategies to use a combination of different attributes found in a `VirusTotal` scan report to label the apps as malicious and benign. In contrast, random forests with decision trees of depth one are incapable of accurately labeling apps, especially if they use scan report attributes other than the verdicts of a `VirusTotal` scanner.

6.2.2. Naive Features

Using naive features and grid search `Naive GS`, we found that the random forests that make up the ML-based labeling strategies had three different structures and depths. The random forest trained using this type of feature on November 2018 had depths of ten checks. For readability, in Figure 6.3, we show parts of an example decision tree that belongs to this random forest. The decision tree in the figure checks the verdicts of different `VirusTotal` scanners, some of which we found to be correct in Section 5.2, such as `Fortinet` and `Ikarus`, along with others, such as `Cyren`. However, checking the verdicts of these different scanners yield pure leaf nodes that are confident about the class of an app (i.e.,

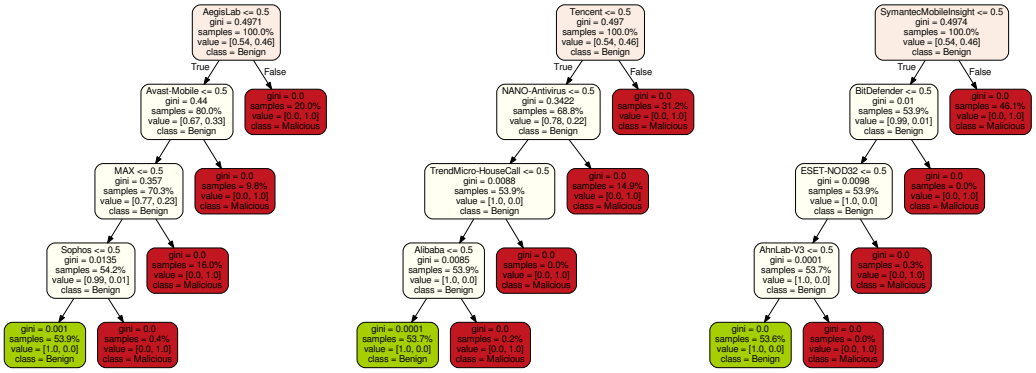


(a) Using **all** naive features (b) Using **selected** naive features

Figure 6.6.: Two decision trees trained using grid search and naive features `Naive GS` and `Naive Sel GS` extracted from apps in the *AMD+GPlay* dataset in November 2018.

Gini index of zero). Although this random forest had decent and stable MCC scores on both test datasets, one can notice the presence of checks similar to the `Babable` example with engineered features, namely checks that rely on the absence of a scanner's verdict. For example, the fourth node and its two child nodes check whether the verdicts of the scanners `Jiangmin`, `VIPRE`, and `Kingsoft` do not exist in the scan report being processed.

This type of checks does not exist in decision trees trained using selected naive features. As seen in Figure 6.6b, decision trees in the `Naive Sel GS` random forests had depths of four, and their structures were different from the ones trained with all naive features in two aspects. First, they relied on the verdicts of correct scanners. That is we did not find any decision trees that rely on the verdicts of `VirusTotal` scanners, such as `GData`, `Alibaba`, or `Kingsoft`. Second, they relied on the verdicts of scanners that are included in the scan reports and checked whether each scanner deemed an app as malicious or benign rather than whether the scanner's verdicts were absent from the scan report. The performance of `Naive Sel GS` ML-based labeling strategies mimicked that of their `Naive GS` counterparts on the *Hand-Labeled* dataset. However, on the newer *Hand-Labeled 2019* dataset, that performance decreases in terms of MCC scores by about 0.1, albeit remaining stable.

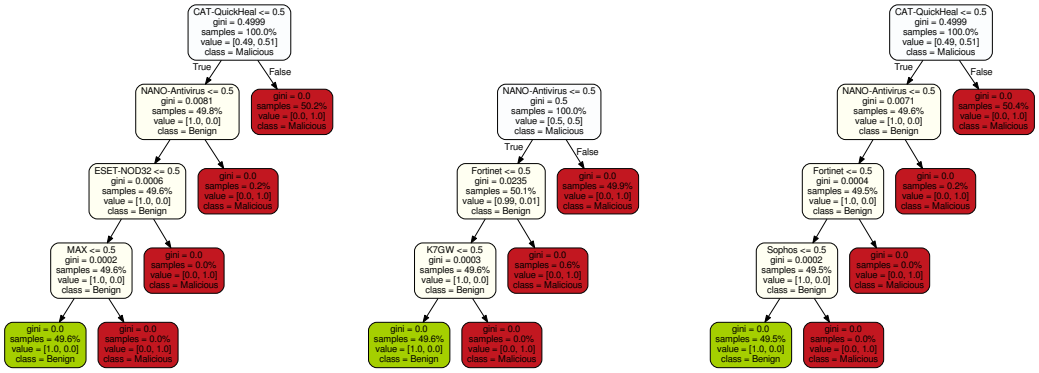


(a) April 12th, 2019 (b) May 10th, 2019 (c) June 7th, 2019

Figure 6.7.: Three randomly selected decision trees in the ML-based labeling strategies' random forests trained using grid search and **all** naive features `Naive GS` extracted from `VirusTotal` scan reports of apps in the *AMD+GPlay* dataset downloaded on April 12th, 2019, May 10th, 2019, and June 7th, 2019.

The depth of four does not seem to hinder the performance of ML-based labeling strategies using naive features. As seen in Figure 6.7 and Figure 6.8, using an average depth of four, the labeling strategies that used both all and selected naive features had MCC scores of such strategies trained at different points in time were (a) stable over time, and (b) matched and in some cases outperformed those of threshold-based labeling strategies using the current optimal threshold. The primary difference between random forests trained using `Naive GS` and `Naive Sel GS` was the utilization of correct and stable `VirusTotal` scanners. On the one hand, `Naive GS` decision trees relied on the verdicts of a mixture of correct scanners (e.g., `Sophos`, `ESET-NOD32`, and `NANO-Antivirus`) and other less reliable scanners to label apps. On the other hand, `Naive Sel GS` exclusively relied on the verdicts of correct and stable scanners, as identified in Chapter 5. In both cases, we noticed that the deeper the decision trees grow, the lower the number of malicious apps that need to be labeled. For example, in all six decision trees in Figure 6.7 and Figure 6.8, by the time the last check is performed, less than 1% of the malicious apps remain to be classified. So, it seems that the checks employed by the decision trees attempt to identify malicious apps first, and the remaining apps are being classified as benign.

As for the performance of both types of ML-based labeling strategies on the test datasets, we noticed that labeling strategies using `Naive Sel GS` had fewer fluctuations in MCC scores over time on both datasets than its `Naive GS` counterpart. Moreover, the mediocre performance of ML-based labeling strategies trained using `Naive GS` on April 26th, 2019 increases upon using selected naive features. The reason for that we found is that using selected naive features (i.e., `Naive Sel GS`), the depth of the random forests increases

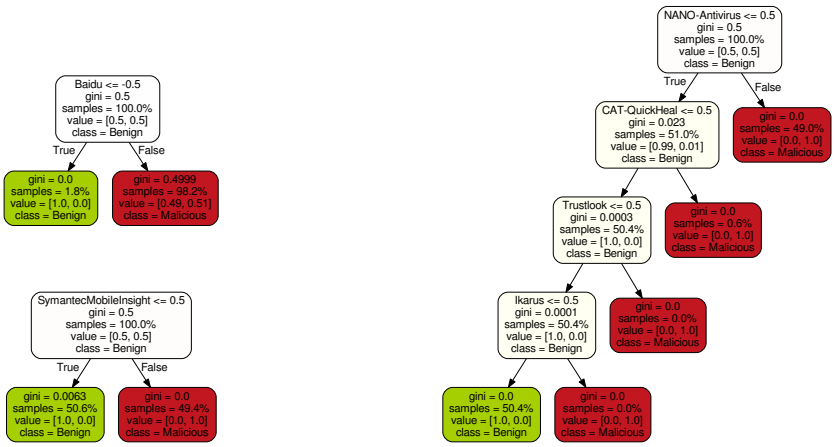


(a) April 12th, 2019 (b) May 10th, 2019 (c) June 7th, 2019

Figure 6.8.: Three randomly selected decision trees in the ML-based labeling strategies' random forests trained using grid search and **selected** naive features `Naive Sel GS` extracted from `VirusTotal` scan reports of apps in the *AMD+GPlay* dataset downloaded on April 12th, 2019, May 10th, 2019, and June 7th, 2019.

from one to four to resemble the structure of other `Naive Sel GS` random forests. We visualize the structure of `Naive GS` and `Naive Sel GS` ML-based labeling strategies trained on this date in Figure 6.9. On the one hand, random forests trained using all naive features (seen in Figure 6.9a) result into decision trees that employ only one check of different `VirusTotal` scanners. The more reliable and correct the scanner is (e.g., `NANO-Antivirus` or `SymantecMobileInsight`), the more confident are the labeling decisions. Decision trees that rely on the verdicts of scanners, such as `Baidu`, are likely to yield unconfident labels. On the other hand, using selected naive features yields random forests with decision trees of an average length of four, which rely on the verdicts of four scanners, most of which we found to be correct and stable.

In summary, we found that using naive features, Maat trains ML-based labeling strategies whose random forests comprise decision trees of depth between three and four. Without feature selection, these decision trees use the verdicts of different `VirusTotal` scanners, some of which were not among the set of correct scanners that Maat identified in Section 5.2. While including the verdicts of those scanners did not significantly decrease the labeling accuracy of ML-based labeling strategies that use them, they may have introduced noisy checks in the strategies' random forests that may have contributed to the fluctuation of their performance especially against the *Hand-Labeled 2019* dataset. Selecting the subset of `VirusTotal` scanners whose verdicts are more informative during training (i.e., using selected naive features), led to training random forests of the same depth but rely exclusively on the verdicts of `VirusTotal` scanners that we found correct in Section 5.2. The performance of ML-based labeling strategies that use selected naive features (i.e., `Naive Sel GS`), is



(a) Using **all** naive features

(b) Using **selected** naive features

Figure 6.9.: Three randomly selected decision trees in the ML-based labeling strategies' random forests trained using grid search and naive features extracted from `VirusTotal` scan reports of apps in the *AMD+GPlay* dataset downloaded on April 26th, 2019. The two trees on the left were trained using **all** naive features `Naive GS` and the tree on the right was trained using **selected** naive features `Naive Sel GS`.

better and closer to that of Best Thresholds labeling strategies and exhibits almost no fluctuations over time on both test datasets.

The insights gained in this section can help us address **(RQ4.3)** as follows:

- We found that naive features train ML-based labeling strategies that have higher and more stable MCC scores on both test datasets than their counterparts trained using engineered features.
- For both types of features, we found that selecting the most informative features boosts the labeling accuracies of ML-based labeling strategies and relatively smoothens the fluctuations of their MCC scores over time.
- We found that decision trees that rely on the verdicts of between three and four `VirusTotal` scanners, which we found to be correct and stable on the training dataset *AMD+GPlay*, yields the best and most stable MCC scores. In fact, we found that the majority of these decision trees correctly label malicious apps after consulting only one of those correct `VirusTotal` scanners. However, the trees are built to consult other `VirusTotal` scanners in case the previous scanner labeled the apps as benign, or its verdict was not included in the scan report (see `VirusTotal`'s first limitation). Effectively, the decision trees label an app as malicious only if one out of three to four correct `VirusTotal` scanners deem them as such. Otherwise, the app is assumed to be benign.
- The structure of Maat's ML-based labeling strategies using selected naive features appears to be more resilient to `VirusTotal`'s dynamicity than threshold-based labeling strategies in general. First, recall that `VirusTotal` sometimes uses inadequate versions of otherwise competent scanners (i.e., third limitation), which causes the verdicts of such scanners to be incorrect, especially against malicious apps. By selecting only the verdicts of `VirusTotal` scanners that are correct, Maat's ML-based labeling strategies can mitigate this limitation. Second, basing the label of an app on the verdicts of between three and four correct `VirusTotal` scanners helps these ML-based labeling strategies yield accurate labels even if the verdicts of one or two of such scanners were not present in the scan report of an app (i.e., `VirusTotal`'s first limitation). Furthermore, since the final label of an app is based on the labels given by 100 decision trees using different combinations of `VirusTotal` scanners, the likelihood of this aspect of `VirusTotal`'s dynamicity to impact the decision of enough trees to lead to an incorrect final decision is low.

To conclude, relying on the verdicts of a three to four `VirusTotal` scanners appear to help Maat's ML-based labeling strategies achieve high labeling accuracy that is steady over a period of time between seven and 12 months, as discussed in the previous section. Furthermore, the structure of decision trees in the random forests that constitute these ML-based labeling strategies appear to be resilient to some aspects of `VirusTotal`'s dynamicity. However, as seen in Section 6.1 and in this section, the structures of Maat's

ML-based labeling strategies differ depending on the utilized feature set and the scan date on which the strategies were trained. We hypothesize that such differences are due to VirusTotal's dynamicity. In the next section, we attempt to identify the aspects of VirusTotal's dynamicity that impact the structure of the trained random forests of Maat's ML-based labeling strategies, effectively addressing **RQ5.1**.

6.3. Sensitivity to VirusTotal's Dynamicity

We hypothesized that the difference in the structure of some ML-based labeling strategies using different feature sets (i.e., engineered versus naive), along with the fluctuations of their labeling accuracies, may be due to the dynamicity of VirusTotal. In this section, we attempt to find out whether VirusTotal's dynamicity has an impact on the structure and labeling accuracy of Maat's ML-based labeling strategies (i.e., **(RQ5.1)**).

6.3.1. Impact of VirusTotal's Dynamicity During Training

In Section 6.1, we mentioned that Maat uses the technique of grid search to find the hyperparameters that yield the most accurate random forests that constitute Maat's ML-based labeling strategies. One of those hyperparameters controls the maximum allowed depth of all decision trees in the random forest (i.e., *max_depth*). We varied the value of *max_depth* to be one, four, ten, and None (i.e., no limitation), and allowed the technique of grid search to identify the value that yields the best validation accuracy ($\frac{TP+TN}{P+N}$) achieved using the technique of cross-validation, which we set to be ten-fold. In our case, this accuracy is calculated by splitting the scan reports of apps in the *AMD+GPlay* dataset into ten folds, train the random forests using nine of those, and using the remaining one-tenth as a validation dataset. The final validation accuracy is an average of all ten accuracies achieved on the ten validation datasets. We found that the validation accuracies achieved by random forests of different depths are very close to one another. For example, for Naive GS ML-based labeling strategies trained on April 26^{th}, 2019, we found that the validation accuracies achieved by random forests with *max_depth* values of one, four, ten, and None were 1.0, 0.9999794703346335, 0.9999794703346335, and 0.9999589490951507, respectively. Based on those values and despite the insignificant difference in validation accuracies, the grid search algorithm suggested the random forests with *max_depth* of one as the best random forest on April 26^{th}, 2019. As discussed in the previous sections, a depth of one does not help the ML-based labeling strategies to accurately label apps, especially newly-implemented ones (e.g., the apps in the *Hand-Labeled 2019* dataset), which is apparent in the labeling accuracies of the ML-based labeling strategies trained on the aforementioned date. We experimented with forcing the *max_depth* value to be as high as 10. The results obtained from such decision trees were indeed better than those obtained from their counterparts of depths one. However, they did not perform better than decision trees with depths of three and four, whose validation accuracies were the highest.

As discussed in Section 5.5, we implemented Maat in a deterministic manner so that using the same VirusTotal reports would yield the same random forest. Thus, the only variable that would cause two random forests trained using the VirusTotal scan reports of apps in the same dataset (i.e., *AMD+GPlay*), but on two different dates, say April 12^{th} 2019 and April 26^{th} 2019, to have different *max_depth*s is the content of the VirusTotal scan reports, which differed on both dates. We already discussed that VirusTotal frequently changes the set of scanners it includes in the VirusTotal scan reports of apps. This change need not be significant to impact the depths of Maat's ML-based labeling strategies. For example, on April 26^{th}, 2019, the difference between the validation accuracy of random forests with depths one and four was merely 0.00002053. This means that out of one tenth of the total number of apps in the *AMD+GPlay* datasets (i.e., $48,715 \times 0.1 = 4,871$), an average of 0.10 (i.e., $4,871 \times 0.00002053 = 0.100011895$) apps gets misclassified in the validation dataset. Since this value is an average, we can assume that at some validation folds, a few apps out of 4,871 are misclassified. We found that between April 12^{th}, 2019, and April 26^{th}, 2019, only 15% of the apps in the *AMD+GPlay* dataset had the exact same verdicts. Furthermore, between these two dates, almost 85% of the apps had at least one verdict change, 51.65% had at least two verdicts change, and 23.4% had at least three verdicts change. This change of verdicts between the two dates, we assume, might cause a difference in validation accuracies enough to make Maat favor random forests with a maximum depth of four rather than one.

6.3.2. Impact of VirusTotal's Dynamicity During Test

Another aspect of the ML-based labeling strategies' sensitivity to VirusTotal's dynamicity is evident in the fluctuations of their MCC scores over time, especially on the *Hand-Labeled 2019* (i.e., during testing the labeling accuracy of the trained strategies). The frequent change in the scanners used in the scan reports of these new apps affects the attributes extracted from these scan reports (e.g., *positives*), and the verdicts given by scanners. In turn, the performance of labeling strategies that use both types of features is affected. As discussed earlier, using selected features seems to stabilize the performance of ML-based labeling strategies, especially those using naive features. However, on November 8^{th}, 2019, one can notice the decrease in the MCC scores of even the most stable strategies (i.e., Naive Sel GS). In Section 4.3, we demonstrated VirusTotal's limitation of manipulating the scanners in the scan reports of malicious apps in the *Hand-Labeled 2019* dataset, which affected the performance of threshold-based labeling strategies that once yielded the best labeling accuracies. Similarly, since Naive GS and Naive Sel GS labeling strategies solely rely on the verdicts of VirusTotal scanners to label apps as malicious, removing a number of them from the scan reports of apps is expected to impact the decision trees that rely on their verdicts to label apps. For example, given that they are among the correct and stable scanners that Naive Sel GS relies on, removing the verdicts of the ESET-NOD32, Fortinet, and Ikarus scanners from the scan report of the malicious app 90e6ac481fdd497f152234f1cd5bec6d40f50037 will have a negative impact on

decision trees that employ these scanners. Fortunately, the decision trees of such labeling strategies combine the verdicts of different scanners, which keeps the misclassifications to a minimum. That is the highest drop in MCC scores for the `Naive Sel GS` strategies between October 25^{th}, 2019 and November 8^{th}, 2019 is 0.14 which is a decrease of 18%.

To conclude this section, we gained the following insights about the impact of `VirusTotal`'s dynamicity on the structure and labeling performance of Maat's ML-based labeling strategies, effectively addressing **(RQ5.1)**:

- `VirusTotal`'s frequent and unexpected manipulation of the set and versions of scanners it includes in scan reports of (Android) apps (i.e., its first and third limitations), impacts apps that are almost ten years old in the *AMD+GPlay* dataset. Albeit smaller than those encountered by newly-developed apps, these changes cause insignificant changes in the `VirusTotal` scan reports used by Maat to train its ML-based labeling strategies and, in turn, the validation accuracies of random forests using different hyperparameters. We found that these changes might cause Maat to favor random forests with shallow depths (e.g., one), which yield ML-based labeling strategies that are incapable of accurately labeling apps based on their `VirusTotal` scan reports, especially newly-developed ones in the *Hand-Labeled 2019* dataset.
- The same limitations of `VirusTotal` cause the fluctuations in the performance of some Maat ML-based labeling strategies. In the previous sections, we found that using selected naive features stabilizes these fluctuations and helps the ML-based labeling strategies maintain high MCC scores mimicking those of the Best Thresholds (i.e., the upper bound of labeling accuracy). However, similar to other labeling strategies, Maat's ML-based labeling strategies using selected naive features are constrained by the quality of the verdicts in `VirusTotal` scan reports. So, as seen on November 8^{th}, 2019, `VirusTotal`'s removal of the verdicts of three scanners that correctly labeled two malicious apps from their scan reports had a negative impact on the ML-based labeling strategies as well.

6.4. Enhancing Detection Methods

The second criterion we use to evaluate the applicability of Maat's ML-based labeling strategies is their ability to contribute to training more effective ML-based detection methods (i.e., **(RQ4.4)**). In Chapter 1, we discussed that inaccurate labeling of apps used to evaluate detection methods might have a negative impact on their detection performance. By addressing this issue of labeling accuracy, we can contribute to helping the research community focus on developing effective detection methods rather than being consumed by devising labeling strategies that accurately label apps in their training datasets. The main hypothesis we wish to test in this section is that using labeling strategies that yield labels reflecting the ground truth of apps (i.e., malicious and benign), should lead to more effective detection of out-of-sample apps. That is, based on their labeling performance in

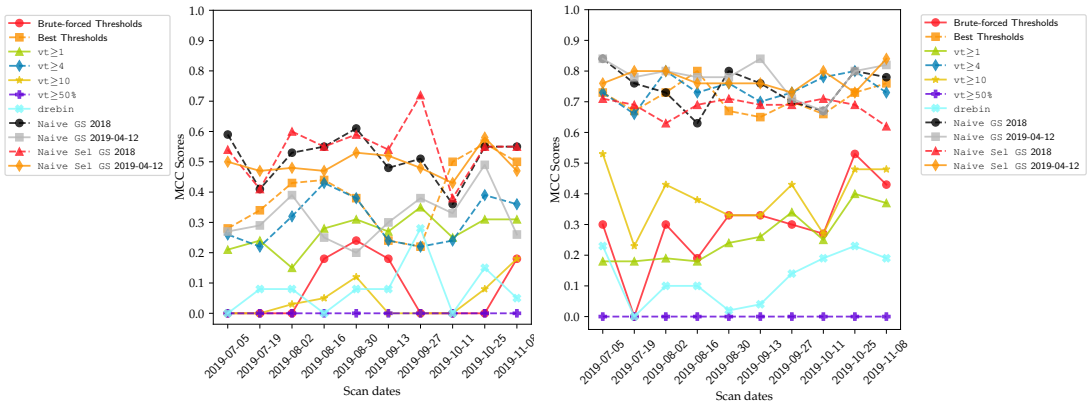
the previous section, we hypothesize that Maat's ML-based labeling strategies–in particular ones using selected naive features–will contribute to training ML-based detection methods that are more effective than ones trained using conventional labeling strategies.

First and foremost, we focus on ML-based detection methods given their popularity within the academic community [100, 139, 142, 156, 16]. In essence, we train different ML classifiers using static features extracted from the *AndroZoo* dataset. The feature vectors are then labeled using different threshold-based and ML-based labeling strategies. We test the detection abilities of such classifiers by assessing their ability to label apps in the *Hand-Labeled*, and *Hand-Labeled 2019* datasets correctly. For readability, we use the shorter term *classifier was labeled* instead of *classifier, whose feature vectors were labeled*.

There are a plethora of approaches to using static features and ML algorithms to detect Android malware. We utilize a detection method that is renowned in the research community and has been used by different researchers as a benchmark [44], namely *Drebin* [15]. The *Drebin* approach comprises three components: a linear support vector machine Linear Support Vector Machine (SVC) to classify apps, a set of static features extracted from Android apps that spans all app components, permissions, URLs, etc., and the `drebin` labeling strategy. In particular, given a dataset of Android APK archives, *Drebin* method collects all component names (e.g., activities), permissions, URLs found in all of those apps and consider each unique occurrence as a feature. For each identified feature, if an app's APK archive contains this feature (e.g., requests particular permissions), the corresponding feature in the app's feature vector will set as 1.0; otherwise, it will be set as 0.0. Using an implementation of *Drebin*'s feature extraction algorithm, we extracted a total of 71,260 features from apps in the *AndroZoo* dataset. Those feature vectors are labeled based on the `VirusTotal` scan reports of their apps using the `drebin` labeling strategy we introduced before (see Section 5.2). Lastly, a SVC model is trained using the training feature vectors (i.e., *AndroZoo*).

In addition to *Drebin*, we use the following classifiers: *KNN* [122], *RF* [123], *SVM* [15], and *GNB* [123]. Given that the KNN and RF classifiers can have different values for their hyperparameters, we used the technique of grid search to identify the classifiers that yielded the best validation accuracies and used them as representatives of those classifiers. Consequently, we refer to these to classifiers as KNN and RF. To train these classifiers, we statically extracted numerical features from the APK archives of apps in the *AndroZoo*, which depict information about the apps' components [124, 123], the permissions they request [15, 154], the API calls found in their codebases [113, 159], and the compilers used to compile them [136].

The labeling strategies we consider in this experiment are the conventional threshold-based strategies of `vt`$\geq$`1`, `vt`$\geq$`4`, `vt`$\geq$`10`, `vt`$\geq$`50%`, and `drebin`, the threshold-based labeling strategies that use thresholds brute-forced at each point in time, threshold-based labeling strategies that use the best threshold at each point in time, and Maat's ML-based labeling strategies that had the highest MCC scores in the previous set of experiments, namely `Naive GS` and `Naive Sel GS`. In assessing the performance of different ML-based detection methods, we are **not** concerned with absolute values that indicate the



(a) *Hand-Labeled* (b) *Hand-Labeled 2019*

Figure 6.10.: The MCC scores achieved by the *Drebin* classifiers labeled using different threshold-/ML-based labeling strategies against the *Hand-Labeled* and *Hand-Labeled 2019* dataset between July 5^{th}, 2019 and November 8^{th}, 2019.

quality of the classifier and the features it is trained with. Instead, we are interested in their performance with respect to one another.

In Figure 6.10 we plot the classification performance of *Drebin* classifiers labeled using different labeling strategies against apps in the *Hand-Labeled* and *Hand-Labeled 2019* datasets in terms of MCC score. Each point on any line depicts a date on which the apps in the training dataset, *AndroZoo*, were re-scanned, and their up-to-date `VirusTotal` scan reports were downloaded. Using the up-to-date `VirusTotal` scan reports of apps in the *AndroZoo* dataset, we label the feature vectors of those apps as malicious and benign with the help of a given labeling strategy; then, a classifier is trained using the labeled feature vectors. After training, the classifier is used to classify the feature vectors of apps in the *Hand-Labeled* and *Hand-Labeled 2019* as malicious and benign. The recorded predicted labels are then compared against the ground truth we generated for apps in these datasets by manually analyzing them, as discussed in Section 1.3.4, and an MCC score is calculated to depict the prediction's accuracy. Given that the lines depicting the MCC scores of different classifiers are intertwined, we tabulate their MCC scores in Table 6.1 for better readability.

Studying these performances, we made the following observations:

1. The first observation we made was that the performance of *Drebin* classifiers is better on the *Hand-Labeled 2019* dataset than on the older *Hand-Labeled* dataset. The reason behind this is that apps in the *AndroZoo* dataset were developed between 2018 and 2019. Probably, features extracted from these apps are expected to be similar to apps in the *Hand-Labeled 2019* dataset more than to those in the *Hand-Labeled* dataset, especially since the *Drebin* feature set is designed to identify similarity in the components and features utilized by the Android apps in the training dataset. This proximity in feature

Table 6.1.: Detailed view of the MCC scores achieved by the *Drebin* classifiers labeled using different threshold-/ML-based labeling strategies against the *Hand-Labeled* and *Hand-Labeled 2019* dataset between July 5^{th}, 2019 and November 8^{th}, 2019.

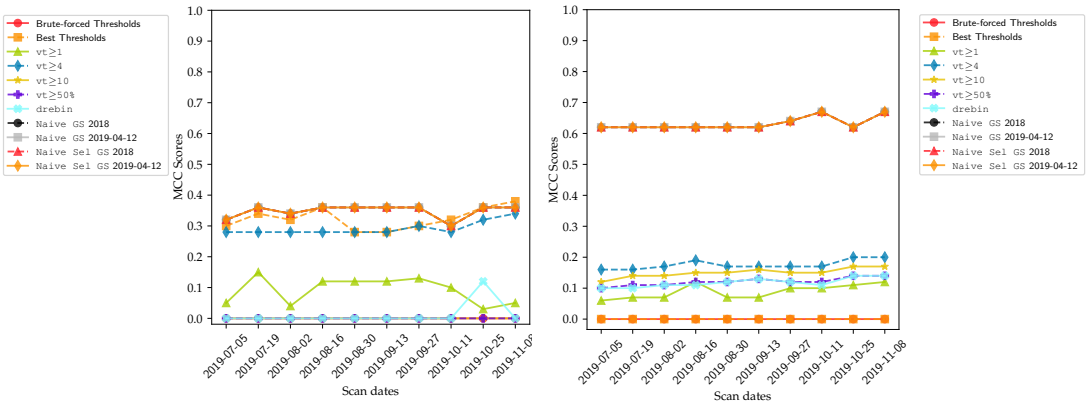
Labeling Strategy \ Scan Date	2019-07-05	2019-07-19	2019-08-02	2019-08-16	2019-08-30	2019-09-13	2019-09-27	2019-10-11	2019-10-25	2019-11-08
Hand-Labeled										
Brute-forced Thresholds	0.00	0.00	0.00	0.18	0.24	0.18	0.00	0.00	0.00	0.18
Best Thresholds	0.28	0.34	0.43	0.44	0.38	0.24	0.22	0.50	0.56	0.50
vt≥1	0.21	0.24	0.15	0.28	0.31	0.27	0.35	0.25	0.31	0.31
vt≥4	0.26	0.22	0.32	0.43	0.38	0.24	0.22	0.24	0.39	0.36
vt≥10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
vt≥50%	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
drebin	0.00	0.08	0.08	0.00	0.08	0.08	0.28	0.00	0.15	0.05
Naive GS 2018	0.59	0.41	0.53	0.55	0.61	0.48	0.51	0.36	0.55	0.55
Naive GS 2019-04-12	0.27	0.29	0.39	0.25	0.20	0.30	0.38	0.33	0.49	0.26
Naive Sel GS 2018	0.54	0.51	0.60	0.55	0.59	0.54	0.72	0.38	0.55	0.55
Naive Sel GS 2019-04-12	0.50	0.47	0.48	0.47	0.53	0.52	0.48	0.43	0.58	0.47
Hand-Labeled 2019										
Brute-forced Thresholds	0.30	0.00	0.30	0.19	0.33	0.33	0.30	0.27	0.53	0.43
Best Thresholds	0.73	0.67	0.73	0.80	0.67	0.65	0.70	0.66	0.73	0.76
vt≥1	0.18	0.18	0.19	0.18	0.24	0.26	0.34	0.25	0.40	0.37
vt≥4	0.73	0.66	0.80	0.73	0.76	0.70	0.73	0.78	0.80	0.73
vt≥10	0.53	0.23	0.43	0.38	0.33	0.33	0.43	0.27	0.48	0.48
vt≥50%	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
drebin	0.23	0.00	0.10	0.10	0.02	0.04	0.14	0.19	0.23	0.19
Naive GS 2018	0.84	0.76	0.73	0.63	0.80	0.76	0.70	0.67	0.80	0.78
Naive GS 2019-04-12	0.84	0.78	0.80	0.78	0.78	0.84	0.71	0.67	0.80	0.82
Naive Sel GS 2018	0.71	0.69	0.63	0.69	0.71	0.69	0.69	0.71	0.69	0.62
Naive Sel GS 2019-04-12	0.76	0.80	0.80	0.76	0.76	0.76	0.73	0.80	0.73	0.84

space, as discussed in Section 3.4.4, helps *Drebin*'s SVC classifier identify patterns shared by malicious and benign apps better.

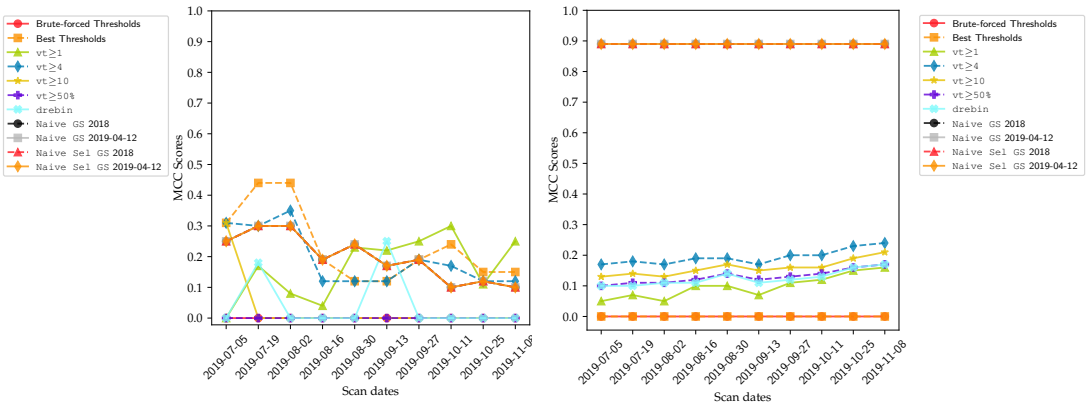
2. Secondly, we found that using best threshold at each point on time along with Maat's ML-based labeling strategies helps the *Drebin* classifier achieve better MCC scores than threshold-based labeling strategies that use fixed thresholds over time (e.g., `vt`$\geq$`1`, `vt`$\geq$`10`, and `vt`$\geq$`50%`), and better than the Brute-forced Thresholds. Moreover, the average MCC scores of *Drebin* classifiers labeled using Maat's ML-based labeling strategies is higher than that of classifiers labeled using the Best Thresholds at each scan date. As seen in Section 6.1, Maat's ML-based labeling strategies using (selected) naive features and the Best Thresholds labeling strategies achieved the highest and steadiest MCC scores that depict their labeling accuracies.

3. The results in Table 6.1 show that using labeling strategies that had the highest MCC scores for labeling accuracy contribute to training ML-based detection methods that achieve the best MCC scores, which supports the hypothesis that the more accurate and reliable a labeling strategy is, the more effective are the detection methods that rely on apps labeled by such a strategy. However, we noticed a number of exceptions that contradict this hypothesis. For example, the `Naive GS` ML-based labeling strategies trained in November 2018 and the `Naive GS` strategies trained on April 12^{th}, 2019 had almost the same labeling accuracy against apps the *Hand-Labeled* dataset (see Figure 6.1). Despite this similarity, the *Drebin* classifiers labeled using the latter strategy noticeably underperforms in comparison to the former one. This behavior switches on the *Hand-Labeled 2019* dataset: the performance of *Drebin* classifiers labeled using `Naive GS` 2018 almost mimics that of those labeled using `Naive GS` trained on April 12^{th}, 2019, despite the former being less able to label apps in this dataset accurately.

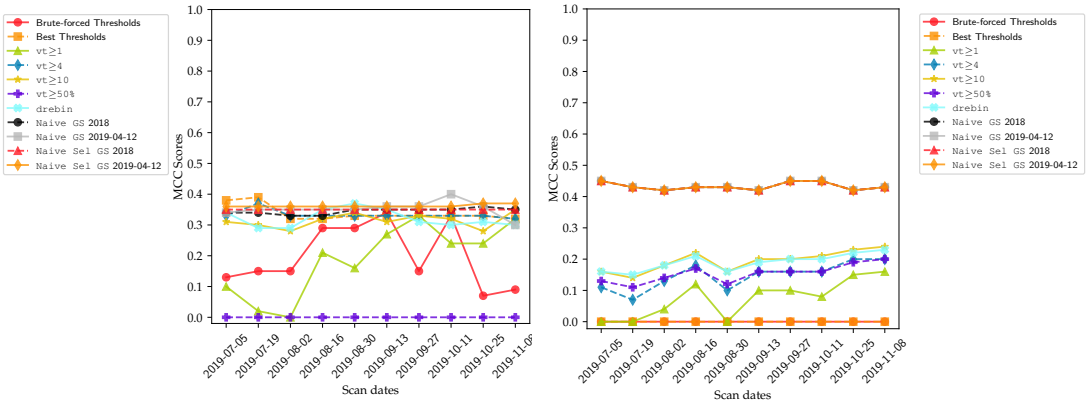
In general, we noticed that MCC score of each *Drebin* classifier differs depending on the utilized labeling strategy and the scan date of the `VirusTotal` scan reports this strategy uses to label apps. In addition to these two factors, we noticed that the utilized classifier also impacts these scores. Figure 6.11 shows the MCC scores achieved by the KNN, RF, and GNB classifiers against the same test datasets using the same labeling strategies. Similar to the *Drebin* classifier, the MCC scores achieved by these three classifiers show that Maat's ML-based labeling strategies contribute to training more effective detection methods than their threshold-based counterparts, in spite of using a different set of features to represent the apps in the training and test datasets. However, unlike the *Drebin* classifier, the MCC scores of the KNN, RF, and GNB classifiers seem to be stable across different scan dates, especially on the *Hand-Labeled 2019* dataset. The values of these scores seem to differ from one classifier to another. On the *Hand-Labeled 2019* dataset, for instance, the average MCC scores of the KNN, RF, and GNB classifiers were 0.64, 0.89, and 0.43, respectively. So, it seems that the features used to represent apps in the training dataset and the type of ML classifier also impact the performance of a detection method. This helps us to



(a) KNN with *Hand-Labeled* (b) KNN with *Hand-Labeled 2019*



(c) RF with *Hand-Labeled* (d) RF with *Hand-Labeled 2019*



(e) GNB with *Hand-Labeled* (f) GNB with *Hand-Labeled 2019*

Figure 6.11.: The MCC scores achieved by the KNN, RF, and GNB classifiers labeled using different threshold-/ML-based labeling strategies against the *Hand-Labeled* and *Hand-Labeled 2019* dataset between July 5^{th}, 2019 and November 8^{th}, 2019.

address **(RQ4.4)** as follows. The results of our experiments show that Maat's ML-based labeling strategies that use (selected) naive features, which proved in Section 6.1 to provide the highest MCC scores in labeling accuracy, contributes to training different ML-based detection methods that are more effective at classifying apps in the *Hand-Labeled* and *Hand-Labeled 2019* dataset as malicious and benign based on vectors of static features extracted from their APK archives than detection methods labeled by conventional threshold-based labeling strategies currently adopted within the research community.

6.5. Summary

We implemented Maat as an attempt to contribute to standardizing the utilization of VirusTotal to label apps used to train and evaluate ML-based labeling strategies as malicious and benign. In Chapter 5, we hypothesized that the ML-based labeling strategies trained by Maat are built to be more resilient to VirusTotal's dynamicity than threshold-based labeling strategies and can, hence, act as a reliable alternative to the conventional threshold-based labeling strategies used within the literature. In this chapter, we attempted to verify this hypothesis by extending **(RQ4)** and **(RQ5)** postulated in Section 1.3.1. Firstly, we found that Maat's ML-based labeling strategies–especially those using selected naive features–can maintain a steady labeling accuracy that matches the upper bound of accuracy achieved by the currently optimal threshold of VirusTotal scanners for a period between seven and 12 months. This means that such labeling strategies need not be re-trained as frequently as the currently optimal threshold needs to be re-calculated due to VirusTotal's dynamicity. Having mentioned VirusTotal's dynamicity, we found, secondly, that the platform's dynamicity may impact the structure of Maat's ML-based labeling strategies and their labeling accuracy. In particular, the frequent manipulation of scanner verdicts and versions might lead to training shallow, random forests, which constitute the ML-based labeling strategies that are unable to yield accurate labels, especially for newly-developed Android apps. Thirdly, we found that different ML-based detection methods whose feature vectors were labeled using Maat's ML-based labeling strategies had better classification scores than those detection methods labeled using conventional threshold-based labeling strategies adopting within the research community. These results imply that Maat's ML-based labeling strategies can be used to label apps based on their VirusTotal scan reports instead of subjective threshold-based labeling strategies that rely on fixed thresholds of VirusTotal scanners, which makes them susceptible to VirusTotal's dynamicity. However, as we discuss in Section 9.3, our results are confined to the datasets we used, the period within which the VirusTotal scan reports of apps in those datasets were downloaded, the feature vectors we extracted from those apps, and the ML algorithms we utilized.

7. An Alternative to VirusTotal

This chapter recaps the limitations of VirusTotal that were unveiled throughout the thesis and details the design of an alternative platform that tackles the issues of VirusTotal, effectively providing the research community with a more reliable, consistent, and stable source of labels for Android apps. Parts of this chapter have previously appeared in peer-reviewed publication [116], co-authored by the author of this thesis.

The research community heavily relies on the online platform, VirusTotal, to acquire and label apps. However, throughout this thesis, we demonstrated the limitations of such a platform, which sometimes undermines its usefulness. Among other issues, the main limitation of VirusTotal is its dynamicity and the frequent change it introduces to its scan reports. In this context, the research community is in need of a more stable and reliable alternative to VirusTotal. In this chapter, we discuss the limitations of VirusTotal (Section 7.1) and use them to propose an alternative platform, designated Eleda, that mitigates those limitations. After detailing the design of such an alternative platform (Section 7.2), we go over its own limitations and the issues that it cannot tackle (Section 7.3).

7.1. A Summary of VirusTotal's Limitations

In this thesis, we identified four major limitations of VirusTotal. Firstly, in Section 4.3, we found that VirusTotal changes the set of scanners it uses to scan apps and includes in their scan reports regularly. In particular, the set of scanners included in the scan reports of some apps seem to randomly change within periods as brief as two weeks. We demonstrated how that might impact the performance of threshold-based labeling strategies by changing the threshold of VirusTotal scanners needed to label apps as malicious, especially if those apps were newly-developed and have immature scan reports. Old Android apps are also affected by this limitation. In Section 5.4, we demonstrated how this regular change impacts the scan report of a malicious Ransom app, which makes it difficult to estimate whether a scan report has stabilized based on attributes such as *positives* and *positives_delta*. Unfortunately, there is no information available that might explain this sudden change in the scanners that VirusTotal uses.

Secondly, although VirusTotal seems to be continuously updating the versions of some of its scanners and claims to be using the latest signature databases, the platform

does not enforce re-scanning of apps. As demonstrated in Section 1.2.1, reanalysis and re-scanning of Android apps have to be triggered manually by the users either via the platform's web-interface or by issuing the proper API requests. In Section 4.4, we show that this design decision significantly delays the process of finding the optimal threshold of VirusTotal scanners to use at any given point in time and may, in fact, render it infeasible. Given that the platform might be hosting millions of benign and malicious apps, re-scanning all apps each time an updated version of a scanner is acquired might be infeasible. However, the platform can regularly re-scan apps every two to four weeks to keep the information in its scan results fresh. Furthermore, although it displays the latest *scan_date* to the users (see Section 3.4.1), VirusTotal does not clearly warn its users of the implications of relying on out-dated scan results.

Thirdly, we found that VirusTotal might change the versions of antiviral scanners to ones that are not designed to cater to Android malware, as detailed in our BitDefender example in Section 5.2. Moreover, we found that VirusTotal uses some antiviral scanners that are not designed for the Android platform, such as McAfee-GW-Edition and Microsoft, which are designed to detected web-based and Windows-based malware, respectively. As mentioned in Section 5.2, the platform claims that antiviral software firms dictate the versions to be used by VirusTotal. Nevertheless, this does not justify using inadequate versions of antiviral scanners to scan apps and including them in their scan reports. We speculate that VirusTotal uses all the antiviral scanners it possesses to scan all apps submitted to the platform regardless of the apps' intended platforms or architectures.

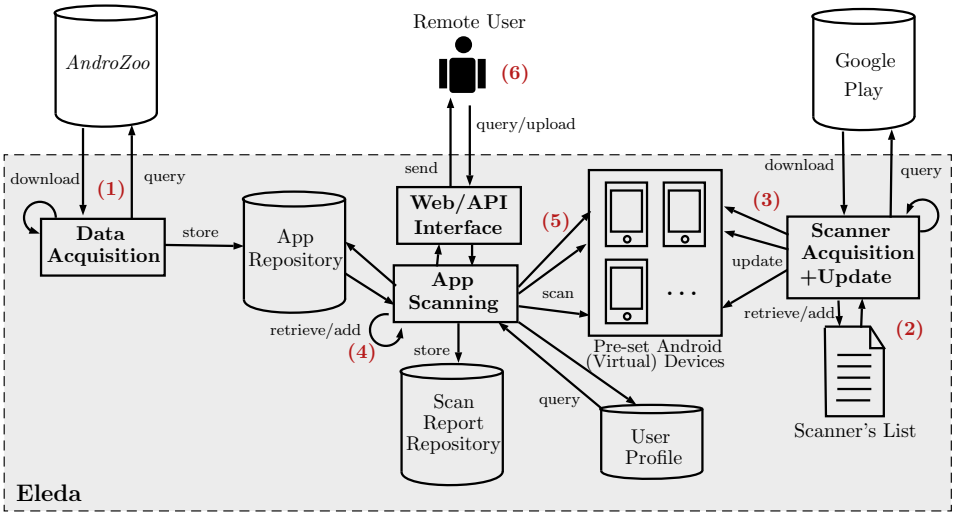
Lastly, having mentioned old scan results, VirusTotal does not offer free access to older scan reports or the scan history of apps. In fact, we are not sure whether such information is available under commercial licenses. Combined with regular re-scan of apps using the same (or similar) set of antiviral scanners, access to older scan report provides researchers with the opportunity to study the evolution of scan reports to estimate, among other things, the time it takes antiviral scanners to detect malicious apps, the amount of time it takes a scan report to stabilize, and whether antiviral scanners change their initial labels.

7.2. Platform Overview

In this section, we discuss how the limitations we discussed above can be addressed. Given that we do not know the internal structure of VirusTotal and how it can be amended to address those limitations, we propose an architecture of a new platform, Eleda[1], that is designed to mitigate VirusTotal's limitations. An overview of Eleda's modules and operations can be seen in Figure 7.1.

[1] Eleda is one of the mirror twins in Sharon Shinn's novel *The Truth-Teller's Tale*, who is a truth-teller incapable of telling lies, earning her the society's trustworthiness. With Eleda's proposed design, we aspire to provide the research community with an alternative to VirusTotal that is more stable and reliable.

Figure 7.1.: An overview of the modules and operations of the proposed VirusTotal replacement, Eleda.



Data Acquisition. The first operation of Eleda is to acquire APK archives of unlabeled Android malicious and benign apps to scan and analyze (step **(1)**). As discussed earlier, *AndroZoo* [10] automates the process of crawling different app marketplaces and continuously downloads their APK archives. Access to such a platform is granted to researchers via an API key that can be used to download apps using cURL. So, using its Data Acquisition module, Eleda can frequently query *AndroZoo* for newly-crawled and downloaded apps, which is indicated using the looped arrow.

Scanner Acquisition and Update. The stored APK archives are meant to be analyzed and, more importantly, scanned. The Scanner Acquisition and Update module is responsible for setting up this scanning infrastructure. In step **(2)**, the module retrieves the names of antiviral scanners and queries app marketplaces, such as Google Play, for their latest versions. This list can be manually populated at first to include the list of VirusTotal scanners that are designed to detect Android malware and are available on Google Play. As of November 2019, 38 (61%) out of around 60 scanners on VirusTotal are available on Google Play. Eleda can also be implemented to add new scanners to the list. After downloading the latest version of each scanner in the list, Eleda updates either an AVD or a physical Android device that is used to scan APK archives (step **(3)**). In general, malware researchers prefer physical Android devices, especially upon analyzing malicious apps that employ anti-virtualization techniques. In this case, Eleda does not attempt to analyze the runtime behavior of Android apps but rather to scan them using a scanning app installed on the (virtual) device. Consequently, there is no advantage of using physical over virtual Android devices. If the downloaded scanner is new and does not have a corresponding

Android (virtual) device, Eleda can create a new (virtual) device either from an Android OS image or from a template of pre-set AVDs and installs the new scanner's APK to this new device. The process of acquiring and updating new versions of antiviral scanners from Google Play–and possibly third-party Android app marketplaces–is meant to mitigate the third limitation of VirusTotal, namely using inadequate scanners and scanner versions not designed to scan Android apps. Moreover, using a pre-populated list of scanners is meant to keep the set of antiviral scanners used to scan APK archives constant (i.e., mitigating the first limitation).

App Scanning To address the second limitation of VirusTotal that it only re-scans Android apps upon request, in step **(4)**, Eleda's App Scanning module retrieves the set of APK archives available in the platform's app repository, scans them using the devices set up by the Scanner Acquisition and Update module, builds the latest scan reports of those apps, and stores them in another repository. In order to make the transition from VirusTotal to Eleda seamless, the scan reports can also be stored in JSON format and can contain the same information contained in VirusTotal's scan reports. For example, static information about the app components can also be extracted using analysis tools, such as *Androguard* [13], and the API calls issued by the app during runtime can also be monitored and recorded [8]. The frequency of the re-scan operation can be set by the users of Eleda. Given that the platform is expected to store a large number of APK archives, it need not re-scan all apps at the same time. Instead, each app can be scanned every constant interval (e.g., two weeks), starting from its initial acquisition date. This setting would reduce the size of the batch of apps that need to be analyzed at once.

User Interaction The last operation of Eleda is user interaction. In step **(6)**, the platform receives a query from a remote user in the format of a hash of an Android app's APK or the archive itself. If the app is not already in the platform's app repository, the App Scanning module can add the app's APK archive to the repository. The APK archive is then scanned using the platform's (virtual) devices, and its scan report can be displayed to the user. Eleda can mimic the design of VirusTotal by offering users to interact with the platform using a web-based interface or using API-requests. However, to address VirusTotal's fourth limitation, Eleda can provide the users with all scan reports of the queried app.

7.3. Challenges and Limitations of Eleda

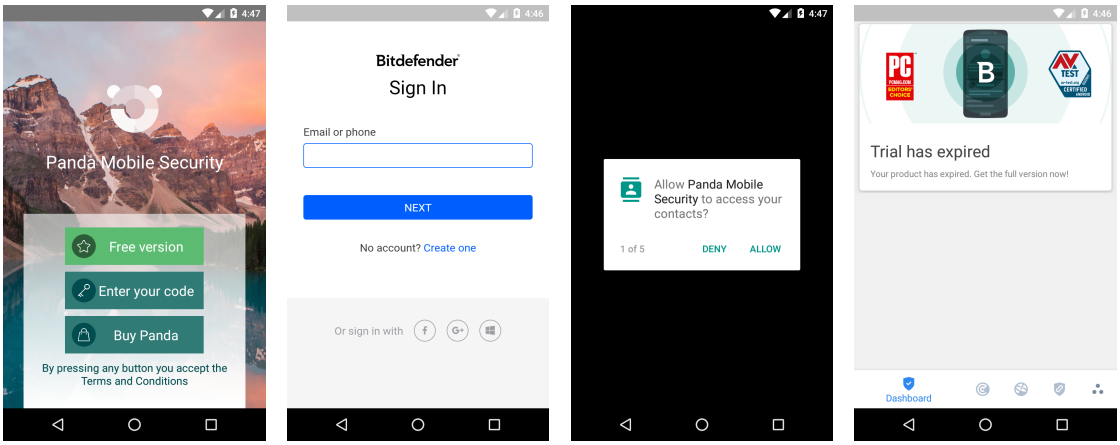
In addressing the limitations of VirusTotal that we mentioned earlier in this chapter, Eleda either introduces new challenges or fails to address the same limitations that VirusTotal suffers from. Similar to VirusTotal, Eleda relies on the verdicts of commercial antiviral scanners. Thus, the correctness of the labels Eleda provides is constrained by the correctness of their labels. With the help of our students, we developed eight malicious Android apps as proofs-of-concept. We did not publish these malicious apps to apps marketplaces, which

might explain why almost all VirusTotal scanners failed to recognize their malignancy, as seen in Appendix F. In other words, since these apps were not encountered in the wild (e.g., app marketplaces), antiviral software did not develop signatures or techniques to detect them. There is no design of Eleda or any other label aggregation platform meant to replace VirusTotal that can mitigate this limitation. The only way to mitigate this limitation is to devise a method that always labels apps correctly, effectively devising a perfect detection method.

The second and third limitations of Eleda are technical and are related to using the commercial versions of antiviral scanners found on the Google Play marketplace. Those versions found on Google Play are designed to be interacted with by humans. This human-centered design means that Eleda might face challenges setting the scanners up and retrieving the verdicts they assign to other apps. As seen in Figure 7.2, commercial antiviral scanners need to be set up before they can be used. On the one hand, the set up process can be as simple as tapping on buttons (Figure 7.2a) or granting permissions (Figure 7.2c). On the other hand, as seen in Figure 7.2b and Figure 7.2d, setting up antiviral apps may include registering to or logging into a service, or more complicated processes to renew a free license or acquire a commercial one. Unfortunately, the setup process differs from one antiviral app to another. So, after downloading and installing the antiviral app (i.e., step **(3)** in Figure 7.1), a human operator needs to set up the app to be ready to scan and label apps. The process of setting up a scanner can be automated using GUI testing tools, such as *Droidbot* [77], which can be configured to perform the sequence of GUI-based events necessary to set up the scanner's app (e.g., enter certain credentials in text fields then tapping a login button). To identify this sequence of events, a human operator still needs to first interact with an app and manually configure a GUI testing tool for future installations of the scanner app.

A similar, yet less complicated, process is that of retrieving the verdicts and labels given by the antiviral scanners install on Eleda's Android devices to apps installed during the App Scanning phase. Given that those verdicts are meant to be displayed to users, the commercial antiviral apps downloaded directly from app marketplaces do not expose any APIs that can be used by Eleda to automatically retrieve the labels they assign to APK archives. So, Eleda needs to parse the notifications displayed on the Android device's screen to retrieve the scanner's verdicts, if any. In Figure 7.3, we give two examples of how Panda's and ESET-NOD32's antiviral apps notify users of the malignancy of apps. Although they can take screenshots of Android devices' displays, GUI-based testing tools are not designed to parse the content of such screenshots.

To the best of our knowledge, most antiviral Android apps scan newly-installed apps and notify users if they found them to be malicious. As seen in the figure, the notification comprises an icon in the top left corner of the device's screen and a notification box laid over all other activities containing details about the detected app and the label the antivirus assigns to it. Eleda can leverage this pattern to scan apps by installing an app's APK archive on an Android (virtual) device and wait for a notification box to be displayed. If a notification box is indeed displayed, the platform can consider that the app has been



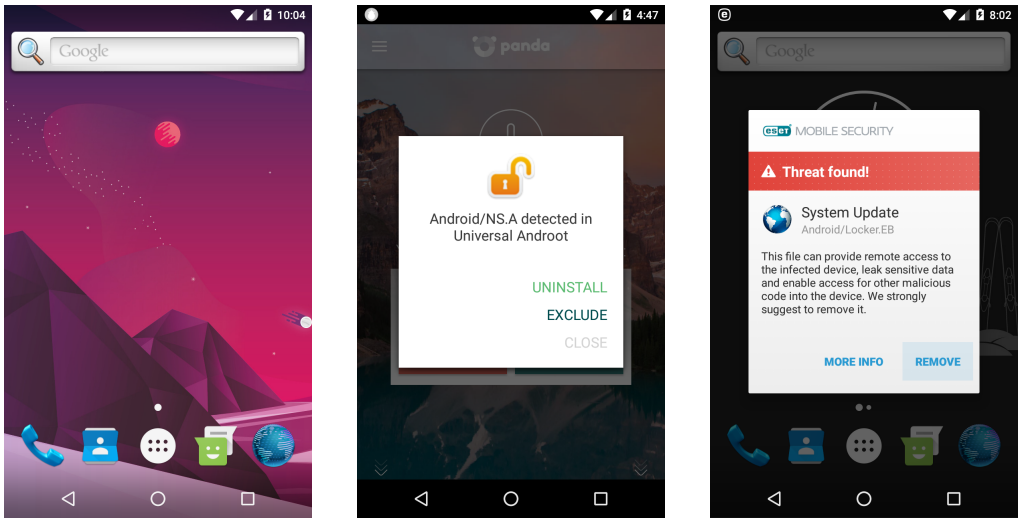
(a) Simple Activation (b) Login/Sign up (c) Grant Permissions (d) License Expiration

Figure 7.2.: Examples of manual operations needed to set up and activate antivirus Android apps downloaded from Google Play.

deemed by the antiviral scanner installed on such a device as malicious.

Technically, this process can be implemented as follows. Eleda can keep a reference of how a neutral display of a template Android device looks like (e.g., Figure 7.3a). After installing an app on an Android (virtual) device, the platform can wait for a pre-defined amount of seconds, take a screenshot of the device's display, and compare the saved screenshot with the reference one. Comparing two screenshots can be carried out by algorithms, such as Structural Similarity Index (SSIM) [152], that compare the structural similarity of two images. If the SSIM measure between the two screenshots is 0.7748 or more, the two screenshots are considered to the same [119], and the app is considered to be benign because the antivirus installed on the device did not display a notification of its malignancy. Otherwise, the screenshot acquired from the device needs to be further processed to ensure that the difference between the two screenshots is due to such a notification. We reckon that an implementation of Eleda can use algorithms to extract text from images [155] and search for strings that indicate the detection of a malicious app (e.g., *detected*, *threat*, and *infected*).

The more challenging task is to extract the malware type and family name from such images. In [55], Hurier et al. discuss the problem of VirusTotal antiviral scanners not following standard procedures to label Android malicious apps. So, malware types and families are not expected to follow a pattern that can be represented using a regular expression, for instance. One possible solution to this problem is to exclude all words found in the screenshot that belong to a natural language. For example, excluding all words that can be found in an English dictionary from Panda's notification (i.e., Figure 7.3b), will leave the strings Android/NS.A and Androot, which can be cross-referenced with pre-extracted malware types and families (e.g., using the *AMD* dataset), to identify the string most likely to correspond to a malware family and/or type. Another solution is to



(a) Neutral Display (b) Panda (c) ESET-NOD32

Figure 7.3.: Two examples of how commercial antiviral apps inform users of the malignancy of installed apps.

customize an extraction method per scanner, given that their notification boxes differ from one another. For example, in Figure 7.3b, the app's family name can be extracted as the string before *detected in Universal Androot*, whereas the family name in Figure 7.3c can be extracted as the string between *System Update* and *This file*. However, the second solution cannot cater to GUI updates in the antiviral scanners.

7.4. Summary

In this doctoral thesis, we identified four limitations in VirusTotal that can undermine the reliability and usefulness of its scan reports. Those limitations can be categorized into (a) VirusTotal's frequent change in the scanners and their versions that it uses to scan Android apps and includes in their scan reports, and (b) the lack of regular scanning of Android apps to build a history of their scans. We propose the implementation of an alternative platform, designated Eleda, that is designed to tackle the limitations of VirusTotal. Needless to say, the proposed features can also be integrated into VirusTotal. The design of Eleda is meant to close the gap between the labels given by the scanners used by VirusTotal and those given by their commercial versions available to users on app marketplaces. Furthermore, the proposed platform is designed to provide the research communities with stable scan reports by adopting the same set of antiviral scanners and only slightly changing them (e.g., in case of a discontinuity). However, unless an agreement is found with antiviral software firms to expose APIs to facilitate Eleda's

tasks, the reliance on commercial versions of antiviral apps introduces limitations to Eleda. While some of those limitations can be mitigated during implementation, some of those cannot be easily tackled, such as setting up antiviral scanners or relaying the inaccurate verdicts of antiviral scanners.

Part III.

Related Work and Conclusion

8. Related Work

This chapter enumerates and discusses related work particularly in the fields of designing and evaluating Android malware analysis and detection methods and identifies research gaps relevant for this thesis. Parts of this chapter have previously appeared in peer-reviewed publications [9], [116], [8], [118], and [119] co-authored by the author of this thesis.

8.1. Defining Malware

There are a plethora of definitions for malware that can be found in dictionaries, in non-scientific articles, and online. As discussed in Chapter 2, such definitions revolve around malware being a subset of apps that jeopardize or undermine the confidentiality, integrity, and availability of a system and the data it contains. In the scientific literature, there are two approaches to defining malware, viz. formal and structural or behavioral.

8.1.1. Formal Definitions

In [68], Kramer et al. devised a general definition for malware as a software system (M) that damages a correct software (S) by directly or indirectly causing it to become incorrect. They defined the correctness of (S) in terms of its intended behavior, which differs from one software system to another. However, they argue that the correctness of a system can be verified using the techniques of model-checking or equivalence-checking, which they abstract as the function $correct(s)$ that checks whether a system is correct. Given a correct system (S), Kramer et al. argue that malware causes the incorrectness of such a correct system by modifying the system itself (e.g., its codebase), the operating system on which it runs, or the data it interacts with [68]. In more abstract terms, Kramer et al. consider a system as malicious if it modifies a pre-defined, verifiable state within which a benign system (S) exists. Effectively, Kramer et al. focus on the integrity aspect of malware.

Stewin et al. focused on a specific type of malware that targets Direct Memory Access (DMA) mechanisms [138]. First, they define 16 classes of DMA-based code that span the combinations of four criteria: (C1) implements malware functionality, (C2) needs physical access to increase the probability of stealthy infiltration, (C3) applies rootkit or stealth capabilities during runtime, and (C4) can survive reboot, standby, or power off modes [138]. Each of these criteria is represented using one bit; a value of zero indicates that the

DMA-based code does not fit the criteria, whereas a value of one indicates that the code does fit it. In total, the 16 classes are represented using four bits. Using this representation, the authors defined DMA malware as DMA-based code that fulfills at least the criteria (C1), (C2), and (C3) (i.e., having the binary representations of either `1110` or `1111`).

For Android malware, we could not find literature that formally defines it. Instead, the majority of work that we found defines malware in terms of its structure and the functionalities it delivers.

8.1.2. Structural and Behavioral Definitions

Definitions of malware that focus on its structure and the tasks it accomplishes date back to Von Neumann's abstract definition of automata (i.e., programs in more recent terms), that *replicate* themselves [149]. Similarly, Cohen defined malware, particularly computer viruses, as programs that infect other benign programs by copying their malicious code into the codebase of their targets, effectively attaching themselves to benign programs [29].

In terms of Android malware, the first study that investigated the structures and functionalities of Android malicious apps on a large scale came four years after its initial release in 2008. In such a study, Zhou and Jiang characterized 1,260 Android malicious apps in terms of their installation methods, activation mechanisms, and the functionalities their payloads deliver [161]. The authors found that around 86% of the apps they analyzed were repackaged versions of benign apps, 36.7% of the apps employ privilege escalation exploits, and 45.3% of the apps generate a monetary profit using premium SMS services [161].

Lindorfer et al. introduced a system, *ANDRUBIS*, that automatically analyzes Android apps [80]. In the process of evaluating *ANDRUBIS*, the authors gathered over one million Android apps. Some of the insights the authors gained about Android malware during this study is that malicious apps (a) request more permissions even if they are not needed, (b) register to more system events via `BroadcastReceivers`, (c) use more third-party advertisement libraries, (d) leak more information about the device (e.g., `IMEI`), over the network, and (e) use the techniques of DCL, reflection, and utilization of native libraries more than their benign counterparts [80].

In [108], Rasthofer et al. focus on a malware family, `Android/BadAccents`, that targeted Korean Banking apps in 2015, and describe the techniques malicious apps in this family used to carry out their functionalities. The authors found that `Android/BadAccents` exploits a *tapjacking* vulnerability in the Android OS to overlay the GUI of particular banking apps and hijack users' credentials [108]. Furthermore, similar to the findings of Lindorfer et al.'s study, the authors found that this family registers to `BroadcastReceivers` to steal incoming SMS messages and abort phone calls. Lastly, by exploiting a zero-day vulnerability to gain administrative privileges, apps in this family remove a particular antiviral scanner (i.e., `AhnLab-V3`), and install a fake one. Rasthofer et al. also reported on the structure of apps in this family and their utilization of a modular architecture that separates the classes carrying out the different aforementioned functionalities.

Li et al. focused in their 2017 study [75] on repackaged, or piggybacked, malware.

They gathered 1,497 benign apps and their repackaged, malicious versions and performed measurements similar to the ones performed in [161] and [80]. Among the multitude of insights Li et al. gained from their measurements, they found that Android repackaged malware requires more permissions than benign apps, reuses the malicious code injected into the benign apps' codebases, and makes use of DCL and reflection.

In [142], Tam et al. surveyed the literature to study the evolution of Android malware analysis and detection techniques. They dedicated part of their work to studying the traits, structures, and behaviors of Android malware as reported by previous studies. That is, they did not reveal new behaviors in Android malware themselves. However, they highlighted malware types and families, such as `Android.hehe`, that uses techniques to recognize its execution within (virtual) analysis environments in order to masquerade its behavior [142].

Wei et al. released a dataset of 24K malicious Android apps, known as *AMD* [153]. During the process of gathering and analyzing those apps, Wei et al. gained the following insights. Firstly, they found that repackaging as a method of distributing malware is declining. That is to say, malware authors increasingly prefer to implement their malicious apps from scratch rather than grafting benign apps with malicious payloads. Secondly, they found that malicious apps increasingly rely on techniques to delay and schedule the execution of their malicious payloads. Thirdly, the authors found that malware authors invest in techniques that prolong the existence of malicious apps on users' devices (i.e., persistence). Similar to previous studies, Wei et al. reported that Android malware continuous to use anti-analysis and detection evasion techniques, such as DCL, reflection, obfuscation, and environment fingerprinting. Lastly, they found that malware authors rely less on premium SMS services as a source of monetary profit in favor of banking malware. Nevertheless, the majority of apps they gathered from the wild comprise `Adware` apps.

A more recent study that provides an overview of the common behaviors exhibited by Android malware families was conducted by Suarez-Tangil et al. in 2018 [140]. In this study, the authors analyzed 1.2 million Android malware samples gathered over a period of eight years, making the study the largest to date. Similar to Li et al. [75], the authors of this study focused on the malicious behaviors injected into benign apps as part of the repackaging process. Unlike Wei et al. the authors found a concept drift in the threats of malicious behaviors. For example, they found that 40% of the malicious families that date back to 2013 utilized premium SMS services to generate profit, which dropped to 10% in families developed in 2016 [140]. Similarly, the use of native support has increased from 15% in 2011 to 80% in 2017. As for detection evasion techniques, Suarez-Tangil et al. found that about 90% of malicious behaviors found in 2017 instead of about 25% in previous studies. Lastly, the authors estimate that the development of standalone malware in favor of repackaging has increased, albeit they estimate the percentage of standalone malware to be 13% rather than the 35% reported by Wei et al. in [153].

The latest study we found in the literature about the structures and behaviors of Android malware is by Salem and Pretschner [119]. In this study, the authors analyze apps drawn from three datasets released as part of the studies by Zhou and Jians [161], Li et al. [75], and Wei et al. [153]. Comparing the malware families and types found in those three datasets,

they found that (a) malicious behaviors injected into benign apps usually generate monetary profit via advertisements, and (b) in general `Adware` is the most Android malware type. Moreover, using the technique of compiler of fingerprinting, they found that Android malware is increasingly shifting towards being standalone, which supports the findings in [140, 153]. However, using this same technique, they reveal that the definition of repackaging is ambiguous. In particular, they found that 52% of the apps gathered by Zhou and Jiang in [161] are compiled using compilers that imply the usage of source code and IDE's rather than the reverse engineering tools known to be commonly used during the repackaging process, which contradicts the findings in [161] that 86% of the gathered apps were repackaged.

8.1.3. Summary

Formal definitions of malware are usually rare and Android malware is no exception. In the literature, we found that researchers *define* malware in terms of the structures of the malicious apps, the functionalities they implement, and the behaviors they exhibit during runtime, all of which we found to evolve over time. The lack of clear definitions of (Android) malware is perhaps the reason behind the subjectivity of deeming apps as malicious, which is reflected in the different verdicts given by antiviral scanners to the same app. In this thesis, we do not attempt to define Android malware; instead, we attempt to understand the reasons behind the disagreements among different antiviral scanners. Without access to information about how antiviral scanners analyze and label apps, we hypothesized that the subjectivity of malware labeling is due to malignancy being a matter of perspective that varies from one stakeholder to another in the Android ecosystem.

8.2. Malware Datasets

During the decade of researching Android malware, a multitude of datasets comprising Android (malicious) apps has been released by members of the research community. After surveying the literature, we found that such datasets are released under two conditions viz., either as part of a study of Android (malicious) apps and the trends they exhibit or as part of introducing a new method to detect Android malware. Given the existence of hundreds of analysis studies and detection methods, there are a plethora of datasets that we cannot discuss in this thesis. So, we focus on large and renowned datasets that were systematically gathered rather than randomly downloaded from an online repository.

In their 2012 study [161], Zhou and Jiang gathered a total of 1,260 Android malicious apps that were developed between August 2010 and October 2011 and released those apps to the research community, effectively providing researchers with the first and largest collection of Android malware called *Malgenome*. Zhou and Jiang focused on malicious apps that were discovered and analyzed by researchers in industry and academia and acquired them by either downloading the apps from app marketplaces, including Google Play, or by

requesting them from the aforementioned researchers [161]. To ensure that the acquired apps are malicious, the authors relied on the verdicts given by four antiviral scanners, namely `AVG Antivirus Free`, `Lookout Security & Antivirus`, `Norton Mobile Security Lite`, and `Trend Micro Mobile Security`. In particular, they downloaded the latest versions of those scanners from the Google Play marketplace, installed each scanner on a separate AVD, and sequentially installed the acquired apps on each of the AVDs, observing the verdicts given by the four scanners to each app. Although not all scanners managed to recognize the malignancy of all apps, collectively, the scanners deemed all apps as malicious. So, effectively, Zhou and Jiang used a labeling strategy that requires only one scanner–out of the four they used–to deem an app as malicious in order to label it as malicious, which we consider a special case of `vt`$\geq$`1`.

Arp et al. developed a ML-based method to detect Android malware based on features statically extracted from their APK archives and called it *Drebin* [15]. To evaluate the effectiveness and applicability of *Drebin*, they gathered a total of 129,013 Android apps comprising 123,453 benign apps and 5,560 malicious apps. This dataset, widely known as the *Drebin*, contains apps developed between August 2010 and October 2012 that were gathered as follows. Firstly, the authors of the *Drebin* dataset acquired a total of 131,611 apps from different marketplaces, websites, security blogs. Furthermore, the 5,560 malicious apps in this dataset include the 1,260 apps of the *Malgenome* dataset. To label apps in this initial dataset, Arp et al. devised a label strategy, which we referred to as the `drebin` strategy throughout this thesis, that deems apps as malicious according to their `VirusTotal` scan reports if at least two out of ten scanners they focus on deem apps as such. Using those labels, they excluded apps that had labels indicating that they belong to the `Adware` malware type. Similar to Zhou and Jiang, Arp et al. released the *Drebin* dataset to the research community, which replaced the *Malgenome* dataset as the de facto evaluation dataset for Android malware detection methods. In fact, after its discontinuation in December 2015, the *Drebin* dataset served as the source of *Malgenome* dataset, given that it already contains those apps.

As discussed in this thesis, Android apps, including malicious ones, continuously evolve, which renders datasets obsolete within a few years. To provide the research community with an up-to-date source of Android (malicious) apps, Allix et al. implemented an online platform, called *AndroZoo*, that frequently crawls official and third-party app marketplaces to acquire new apps [10]. As of the time of writing this thesis, *AndroZoo* platform hosted more than ten million Android apps. Apps that are newly-discovered as automatically indexed and added to a Comma-Separated Values (CSV) file, called *latest.csv*, that contains metadata about the app and its APK archive. Some of this metadata includes the data on which the APK archive was last modified (*dex_date*), the package name (*pkg_name*), the number of `VirusTotal` scanners that deemed the app as malicious (*vt_detection*), the date on which the app was scanned on `VirusTotal` (*vt_scan_date*), and the marketplaces on which the app can be found (*markets*).

Despite not being a dataset per se, *AndroZoo*'s continuous discovery of new apps made it the main source of acquiring Android apps, and some datasets were built on top of

Table 8.1.: A summary of the datasets we utilize in this thesis, their composition, their sources, and the experiments within which they are used.

Dataset Name	Total Apps	Source	Usage
AMD+GPlay	24,553 (malicious)	*AMD*'s Website	◦ Training ML-based labeling strategies (Chapter 5) ◦ Demonstrating degrees of malignancy (Section 2.5)
	24,162 (benign)	*AndroZoo*'s servers	◦ Calculating scanner correctness (Section 5.2) ◦ Estimating scan report stability (Section 5.3)
AndroZoo	6,172 (unlabeled)	*AndroZoo*'s servers	◦ Training ML-based detection methods (Chapter 6)
Hand-Labeled	100 (76% benign+24% malicious)	*AndroZoo*'s servers	◦ Testing accuracy of labeling strategies (Chapter 4 and Section 6.1) ◦ Testing accuracy of ML-based detection methods (Section 6.4)
Hand-Labeled 2019	100 (90% benign+10% malicious)	*AndroZoo*'s servers	◦ Testing accuracy of labeling strategies (Chapter 4 and Section 6.1) ◦ Testing accuracy of ML-based detection methods (Chapter 4 and Section 6.4)
	(malicious)		

it. For example, apps in the *Piggybacking* dataset built by Li et al. were acquired from *AndroZoo* [75]. Apps in this dataset comprise 1,497 presumably benign Android apps and their malicious, repackaged versions. To label apps as malicious and benign, the authors downloaded the `VirusTotal` scan reports of those apps and deemed apps as malicious if at least one scanner labels them as such and as being, otherwise.

As part of their study [153], Wei et al. released a dataset of 24,650 malicious apps that span different malware families and types and called it the *AMD* dataset, short for android malware dataset. Initially, the authors acquired 1,464,590 unlabeled apps from Google Play, VirusShare, the *Malgenome* dataset, and third-party security companies. Using `VirusTotal` scan reports of those apps, they considered only apps that have 50% of the scanners deeming them as malicious, which brought the number of apps to 52,520 apps. To cluster those apps into families, the authors applied an algorithm to extract the dominant keyword for each app out of the labels given to it by the `VirusTotal` scanners. This process led to grouping apps by malware types, such as `Adware`, `Backdoor`, `Ransom`, et cetera. To further group apps by family name, the authors used an in-house developed clustering algorithm [78] to cluster apps into 71 families and 135 varieties. Lastly, to have a measure of confidence that apps in this dataset are malicious, Wei et al. manually analyzed a random sample of ten apps from each of the 135 varieties. To summarize the result of such a manual analysis, the authors represented the structure and behavior of apps in each malware family variety as a block diagram showing the interactions between different segments of the malicious app and (external) resources (e.g., remote servers). The *AMD* dataset is not the largest Android malware dataset to be released to the research community. However, it is the first Android malware dataset that offered a large amount of metadata about the malicious apps, their types, families, varieties, structures, and behaviors.

As discussed in Section 1.3.4, our primary source of Android malware is the *AMD* dataset. We consider all apps in the dataset as malicious given the relatively thorough process of acquiring and labeling apps employed by its authors in [153]. The largest source of

Android benign and malicious apps we rely on is the *AndroZoo* platform. We acquired a total of 30,535 apps that we use in the *GPlay*, *AndroZoo*, *Hand-Labeled*, and *Hand-Labeled 2019* datasets.

8.3. Studying `VirusTotal`

Given the significant role it plays in the malware analysis and detection process, the research community has studied different aspects of `VirusTotal` and its scanners. In [93], Mohaisen et al. inspected the relative performance of `VirusTotal` scanners on a small sample of manually-inspected and labeled Windows executables. The authors introduced four criteria, called correctness, completeness, coverage, and consistency, to assess the labeling capabilities of `VirusTotal` scanners and demonstrated the danger of relying on `VirusTotal` scanners that do not meet such criteria. The main objective of this study is, therefore, to shed light on the inconsistencies among `VirusTotal` scanners on a small dataset. In [92], Mohaisen and Alrawi built on their previous study and attempted to assess the detection rate, the correctness of reported labels, and the consistency of detection of `VirusTotal` scanners according to the four criteria of completeness, correctness, consistency, and coverage that they previously introduced. They showed that in order to obtain complete and correct (i.e., in comparison to ground truth) labels from `VirusTotal`, one needs to utilize multiple independent scanners instead of hinging on one or a few of them.

Similarly, within the domain of Android malware, Hurier et al. studied the scan reports of `VirusTotal` scanners to identify the lack of consistency in labels assigned to the same app by different scanners [55]. Furthermore, they propose metrics that quantitatively describe such inconsistencies.

In a similar work, Charlton et al. studied the labeling accuracies of different `VirusTotal` scanners and devised an algorithm to measure the relative accuracy between those scanners [24]. Using this algorithm, they rank the `VirusTotal` scanners in terms of trust (i.e., whether their verdicts should be trusted), and concluded that at least four scanners should not be utilized in labeling apps. Effectively, they identified a set of scanners that ought to be used in labeling apps based on their `VirusTotal` scan reports (see discussion in Section 5.2). Dua et al. built on this work in [40] and evaluated five metrics used to assess the quality of `VirusTotal` scanners and their verdicts. In both works, the authors did not tackle `VirusTotal`'s dynamicity and how it impacts the set of `VirusTotal` scanners identified to be correct at one point in time, which we demonstrated in Section 5.2.

More recently, Peng et al. [101] showed that `VirusTotal` scanners exhibit similar inconsistencies upon deeming URL as malicious and benign (e.g., because they are Phishing websites). The authors also showed that some `VirusTotal` scanners are more correct than others, which requires a strategy to label such URLs that does not treat all scanners equally. Furthermore, they show the lack of collaboration between `VirusTotal` and the antiviral firms developing the platform's scanners, which delays labeling URLs submitted

to VirusTotal as malicious.

The closest work to ours is that by Zhu et al. in [162]. Using the VirusTotal scan reports of Windows-based and Android malware gathered daily for a period of almost two years, Zhu et al. attempted to answer the following research questions: (a) when can VirusTotal labels be considered trustworthy? (b) how are VirusTotal scan reports utilized by researchers–according to 115 papers they reviewed–to label apps? And (c) are different VirusTotal scanners equally trustworthy? Their findings are similar to ours discussed in Chapter 4, Chapter 5, and Chapter 6. First, they confirm that VirusTotal scanners do flip their verdicts on a regular basis, which undermines the pursuit of a time period within which the VirusTotal scan report of any given app is expected to stabilize (see Section 4.3, Section 5.3, and Section 5.4). Second, they found that the majority of researchers utilize threshold-based labeling strategies that are heavily impacted by the aforementioned regular flips of VirusTotal scanners. Third, they found that only a group of VirusTotal scanners provide consistently correct verdicts that reflect the ground truth of apps despite the inconsistencies between the desktop and VirusTotal versions of scanners (see Section 5.2).

The aforementioned works of [101, 24, 40, 55, 92, 93] shed some light on the shortcomings of VirusTotal and provide the research community with useful metrics to assess, for example, the accuracy of VirusTotal scanners. In Chapter 5, we build on the insights in [55, 92, 93] by extending the *correctness* criterion introduced in [93] to assess the correctness of VirusTotal scanners over time. However, these works did neither detail the shortcomings of VirusTotal nor demonstrate how they might impact the accuracy of labeling strategies that rely on the information found in the VirusTotal scan reports of apps to label them as malicious and benign. In Section 1.2, we refer to this as the third **gap** in the literature associated with using VirusTotal as a basis of labeling (Android) apps. To address this gap, throughout this thesis, we recorded the labeling performance of threshold-based labeling strategies (Chapter 4) and the verdicts of VirusTotal scanners over time (Chapter 5) in order to identify the aspects of VirusTotal's dynamicity, how it manifests in the scan reports of apps, and the impact those manifestations have on the performance of threshold-based and ML-based labeling strategies. To the best of our knowledge, the work of Zhu et al. in [162] is the only work that provides the research community with insights similar to the ones we discuss in this thesis. Their results seem to be obtained independently and concurrently with ours. The work of Zu et al. is related to ours only in the aspect of studying VirusTotal and its scanners. In addition to studying VirusTotal, its dynamicity, and the impact of this dynamicity on its scanners, our work in this thesis builds on the insights we gain to develop an algorithm (see Chapter 4) to devise threshold-based labeling strategies that can cope with VirusTotal's dynamicity and a framework, Maat (see Chapter 5), that uses VirusTotal scan reports to train ML-based labeling strategies that are more resilient to such a dynamicity for longer periods of time. In the next section, we discuss the works related to such an aspect of our work its contributions.

8.4. Labeling Strategies

Multiple efforts attempt to automate the process of labeling apps according to their VirusTotal scan reports. Surveying the literature, we found that there are two objectives in labeling apps based on their scan reports, viz. unifying their labels and discerning their malignancy.

8.4.1. Label Unification

The lack of universal standards to label and name malicious apps allows different antiviral firms to give different labels to the same malicious app [84, 66]. For example, the same malware can have the labels **Worm:**W32/Downadup.gen!A, Net-**Worm**.Win32.Kido.cp, W32/Conficker.**worm**.gen.a, **Worm:**Win32/Conficker.gen!B, **Worm**-Win32/Conficker.gen!A, which all share the case insensitive substring *worm* [66]. So, considering the problem to be that of string manipulation, one of the main objectives of academic research has been to devise methods to unify those strings into one that still represents the malware type and family of a malicious app.

In [102], Perdisci et al. did not implement a method to unify different antiviral labels. However, they implemented an automated approach, *VAMO*, that assesses methods that cluster similar labels together prior to unifying them. This method can help eliminate noisy labels, which is a prerequisite for the accurate unification of labels. Wang et al. also studied the malware naming discrepancies via analyzing the scan results in [151], and identified two types of such discrepancies, viz. syntactic and semantic. The authors implemented an approach, *Latin*, that considers these two types of discrepancies towards devising a consensus classification of antiviral labels that can be used to look up information about malicious apps in repositories such as *Anubis*. To build such consensus classes, *Latin*, builds correlation graphs of the labels given by antiviral scanners to malicious apps, where the nodes depict the labels and the edges depict the correlation between two labels according to the Jaccard distance between their labels; the graph is then broken into multiple sub-graphs of strongly-connected labels [151]. As a use case, a user can search for reports about a particular malicious app using their own words, which are matched to consensus classes that are bound to different malicious apps.

The work in [102] and [151] did not present tools that output unified labels. In [130], Sebastian et al. presented a fully-automatic, cross-platform tool, *AVCLASS*, that examines different labels given to the same app by different scanners, and returns the most likely family name that such app should assume. For example, if three scanners label an app as Android.Youmi.A (AdWare), ADWARE/ANDR.Youmi.P.Gen, and Adware/Youmi.B, *AVClass* is expected to return a label similar to Adware.Youmi. Given a set of different labels given to a malicious app, *AVCLASS* starts by removing duplicate labels (e.g., Gen:Adware.Solimba.1 and Gen:Adware.Solimba.1␣(B) are considered by *AVCLASS* to be equivalent). The tool then removes suffixes and characters such as dots, commas, colons, semi-colons, et cetera and tokenizes the labels. To remove tokens that do not depict family names, *AVCLASS* converts tokens into lowercase, removes digits at the end of tokens, and

excludes tokens that are smaller than four digits, can be found in an input list of generic tokens or are prefixes of the malware app's hash. Next, using a list of aliases, *AVCLASS* replaces tokens with their aliases and ranks tokens per their occurrences. The token with the highest number of occurrences is considered to be the unified label.

Similarly, Hurier et al. developed a tool, *Euphony*, that mines labels, analyzes the associations between them, and attempts to unify them into common family groups [56]. First, *Euphony* pre-processes labels to derive the family names assigned by antiviral scanners to a malicious app. Second, similar to the work of Wang et al. in [151], *Euphony* builds a correlation graph with nodes depicting the family names (i.e., derived in the first step), given by each antiviral scanner to an app and edges indicating that two antiviral scanners have labeled the same sample with $name_x$ and $name_y$. If two antiviral scanners gave the same family name to an app, the edge would have the value 0.0. According to those weights, the graph is broken into sub-graphs. For each cluster, the family names are ranked per occurrence, and names that have majority votes are chosen as an app's unified label.

In this thesis, we attempt to devise one label for malicious apps based on their `VirusTotal` scan reports, which include multiple labels given by different antiviral scanners. However, we do not attempt to give a label that indicates the malware family or type of the app. Instead, we attempt to devise labels that indicate whether an app is malicious or benign (i.e., a label that discerns the malignancy of the app). The relation between Maat and the previously discussed research efforts lies in the aspect of automating the processing of `VirusTotal` scan reports and their scanners to reach a common label. By mentioning these efforts in this thesis, we wish to shed light on the efforts pre-dating Maat at automatically analyzing `VirusTotal` scan reports.

8.4.2. Discerning Malignancy

A more abstract form of labeling apps is to label them as malicious or benign. Similar to label unification, antiviral scanners do not agree on the nature of apps, which manifests in the verdicts that are given by different `VirusTotal` scanners, as discussed throughout this thesis. Apart from threshold-based labeling strategies, researchers have devised more sophisticated labeling strategies, primarily based on ML.

In [63], Kantchelian et al. used the `VirusTotal` scan reports of around 280K binaries to build two ML-based techniques to aggregate the results of multiple scanners into a single ground truth label for every binary. In the first technique, Kantchelian et al. assume that the ground truth of an app (i.e., malicious or benign), is unknown or *hidden*, making the problem of estimating this ground truth is that of unsupervised learning. Furthermore, they assumed that the verdicts of more consistent, less erratic scanners are more likely to be correlated with the correct, hidden ground truth than more erratic scanners. Thus, more consistent scanners should have larger weights associated with their verdicts. To estimate those weights and, hence, devise an unsupervised ML-based labeling strategy, the authors used an Expectation Maximization (EM) algorithm based on a Bayesian model to estimate those models. The second technique devised by Kantchelian et al. is a supervised one based

on regularized logistic regression. However, the authors did not describe the nature of the features they use to train such an algorithm. So, we assume that they relied on the verdicts of antiviral scanners in a manner similar to the naive features we extracted from `VirusTotal` scan reports, as discussed in Section 5.5.

To devise an automated method to label apps based on different verdicts given by antiviral scanners, Sachdeva et al. [113] performed measurements to determine the most correct `VirusTotal` scanners using scan reports of a total of 5K malicious and benign apps. Using this information, they assign a weight to each scanner that they use to calculate a malignancy score for apps based on their `VirusTotal` scan reports. Depending on manually-defined thresholds, the authors use this score to assign a confidence level of Safe, Suspicious, or Highly Suspicious to test apps.

In the aspect of devising labeling strategies to deem apps as malicious and benign based on their `VirusTotal` scan reports, the closest work to ours is that of Sakib et al. in [114], in which they propose three ML-based models to combine the verdicts of different scanners (e.g., those used by `VirusTotal`), to yield accurate labels that reflect the ground truth of apps. Moreover, based on historical scan results, they devise a method to identify the subset of scanners that are more correct than others over time. Using this method, they estimate the combinations of antiviral scanners that yield accurate labels. Our work is different from this work and the work in [113, 63] providing detailed methods that standardize the interpretation of `VirusTotal` scan reports to label apps and can, hence, be utilized by other researchers to label apps in their own dataset (i.e., the second literature gap). Unlike Maat, those efforts neither provide the technical details necessary to implement their approaches nor provided those approaches as tools or frameworks to be used by other researchers. More importantly, despite reporting decent labeling accuracies, those efforts did not evaluate whether the newly-devised labeling method can indeed help build detection methods and left that for future work: "Improved training labels, as obtained by the techniques presented in this thesis, should result in an improved malware detector." [63]. In this thesis, however, we attempted to evaluate the applicability of Maat by examining the classification accuracies detection methods can achieve using labels predicted by the most accurate threshold-/ML-based labeling strategies. Furthermore, to the best of our knowledge, we are the first to test our labeling strategies against zero-day malware.

9. Conclusions

This chapter summarizes the findings of this thesis and draws conclusions from them to address the research questions posed in Chapter 1. It discusses the limitations of this work and suggests different aspects on how to further enhance it.

Due to the infeasibility of manually analyzing large numbers of apps, the malware analysis and detection research community relies on online platforms, such as `VirusTotal`, to label apps in the datasets they use to train and evaluate their newly-devised ML-based detection methods. The **primary objective of this doctoral thesis** is to provide the malware analysis and research community with methods and techniques to optimally utilize `VirusTotal` to assign labels to Android apps that reflect their ground truth of malicious versus benign. To that end, in this doctoral thesis, we attempted to address three main gaps in the literature that are associated with using `VirusTotal` and its scan reports to label Android apps.

Firstly, the verdicts of scanners in `VirusTotal` scan reports are known not always to be correct. Given that researchers use such scan reports as their source of ground truth, which imposes an upper bound on the performance of their ML-based detection methods, it is imperative to attempt to mitigate the negative impact of some `VirusTotal` scan reports providing inaccurate verdicts by focusing on `VirusTotal` scan reports and scanners that provide reliable verdicts that reflect the malignancy of Android apps. To address this gap, we attempted to identify properties of apps that may contribute to their `VirusTotal` scan reports being less reliable. By analyzing the `VirusTotal` scan reports of 53K Android apps in the *AMD+GPlay* dataset gathered between November 2018 and November 8^{th}, 2019, we found that some malware types have higher detection rates than others (e.g., 0.36 for `Adware` versus 0.61 for `HackerTool`). In pursuit of the set of `VirusTotal` scanners that consistently yield correct labels, we found that `VirusTotal`'s dynamicity hinders identifying a constant and universal set of scanners that constantly assign labels to apps that correctly reflect their ground truth. Furthermore, we found that `VirusTotal` scan reports continue to change years after the development and scanning of an app, which undermines defining stability of a `VirusTotal` scan report in terms of age or a particular attribute. Lastly, we found evidence that `VirusTotal` scanners are oblivious to malicious apps they do not encounter frequently (e.g., in an app marketplace). Thus, home-made malicious apps are likely to be mislabeled as benign for years until they are noticed and analyzed by the antiviral scanners used by `VirusTotal`.

Secondly, in the literature, there are no methods that suggest to the research community

how to standardize the interpretation of VirusTotal scan reports to label apps as malicious and benign, instead of using subjective threshold-based labeling strategies. We tackled this issue in two manners. We studied the labeling accuracy of threshold-based labeling strategies that are widely-adopted within the research community and identified how VirusTotal's dynamicity impacts their performance over time. Based on our findings, we propose avoiding using threshold-based labeling strategies that rely on a fixed threshold to label apps, and devised an algorithm to find the currently optimal thresholds of VirusTotal scanners that would yield labels that reflect the ground truth of apps. However, in order for this algorithm to be effective, some conditions need to be satisfied that cannot be met by all researchers. To complement the aforementioned algorithm and mitigate its limitations, we implemented Maat, a framework that is meant to standardize the process of devising labeling strategies by automating the process of analyzing VirusTotal scan reports to devise ML-based labeling strategies. Using a selected subset of the verdicts of VirusTotal scanners, we found that these ML-based labeling strategies can maintain a labeling accuracy that matches the currently best possible threshold (i.e., upper bound of labeling accuracy), for a period of at least seven months without having to be retrained. Furthermore, using these ML-based labeling strategies to label feature vectors used to train ML-based labeling strategies yield better classification results than other conventional threshold-based labeling strategies.

Lastly, the research community did not sufficiently detail the aspects of VirusTotal's dynamicity, how they manifest in VirusTotal scan reports, and how they can impact the performance of different labeling strategies. The absence of research efforts that detail such aspects and limitations hinders the replacement of VirusTotal with more stable and reliable platforms. Throughout this thesis and with the help of Maat, we managed to identify two aspects of VirusTotal's dynamicity that negatively impact the performance of threshold-based and ML-based labeling strategies, namely the frequent and unexpected manipulation of the set and versions of scanners included in the VirusTotal scan reports of apps. In addition to these two limitations of VirusTotal, we found that the platform does not provide access to the history of scan reports of apps to academic licenses, limits the amount of API requests given to holders of such licenses, which prolongs the processes of re-scanning and downloading current VirusTotal scan reports, and does not automatically re-scan apps to relieve researchers of that burden. Using the identified limitations, we propose a blueprint of an alternative platform, named Eleda, that is designed to mitigate the limitations of VirusTotal.

In this chapter, we will relate our findings in this thesis to the research questions we postulated earlier and answer them. We also discuss the limitations of the implemented method and any threats to the validity of our results and findings. Lastly, we enumerate the future work that can address those limitations and further enhance the method we introduced in this thesis.

9.1. Addressing Research Questions

We postulated research questions that address some issues faced by the Android malware analysis and detection community, particularly the problem of effectively utilizing and interpreting the data provided by the online platform VirusTotal. In addressing such questions, we gained the following insights.

RQ1: How can we deem a VirusTotal scan report of an Android app as stable before using it to label the app?

It is widely assumed within the research community that VirusTotal scan reports tend to stabilize. Given that VirusTotal scan reports and the verdicts of scanners they include are our source of ground truth and, hence, the upper bound of the accuracy of labeling strategies base their verdicts upon those scan reports, we attempted in Section 5.4 to find methods to tell whether a VirusTotal scan report has already stabilized. By identifying whether a VirusTotal scan report is stable, one can avoid unstable VirusTotal scan reports that might contain inaccurate verdicts. With no clear definition in the literature of *stability*, we defined stability in terms of attributes found in a typical VirusTotal scan report (e.g., *positives*). That is, once the values of the chosen attributes cease to change, the VirusTotal scan report can be deemed as stable. However, we found that regardless of the age of the (malicious) apps, its VirusTotal scan reports continue to change, which changes the values of its attributes. So, we may answer this research question as follows:

> **Even after years of first scanning the app on VirusTotal, the platform's dynamicity prevents the attributes in scan reports from maintaining the same values for prolonged periods. Consequently, the stability of a VirusTotal report should be defined as a range of values for some chosen attributes rather than particular values. Unfortunately, choosing the attributes to focus on, the range of values within which they are considered stable, and the period they need to maintain those values is subjective. So, to the best of our knowledge, the respective parameters to tell whether a VirusTotal scan report has stabilized remain to be defined.**

RQ2: What are the properties of an Android (malicious) app (e.g., malware type, age, source), that makes it difficult for VirusTotal scanners to correctly label it?

With no clear way of telling whether a VirusTotal scan report is stable, we attempted to find properties that prevent VirusTotal scanners from correctly labeling apps. Researchers can avoid scan reports of apps that fit these properties and focus on ones whose VirusTotal scan reports are likely to contain more accurate information about the apps' malignancy. In Chapter 2, using the VirusTotal scan reports of apps in the *AMD* dataset, we found that some malware types are less likely to be

recognized as malicious by VirusTotal scanners (e.g., Adware). However, without access to the internal processes adopted by antiviral software companies to label apps as malicious, we cannot pinpoint the reasons behind such differences in detection rates. In Section 5.3, we found that VirusTotal scanners tend to focus on Android apps they often encounter in the wild (e.g., app marketplaces). Regardless of the damage they may inflict on Android devices and users, malicious apps that are rarely encountered by users and scanners are unlikely to be recognized as malicious, as per their verdicts on VirusTotal. Lastly, in Chapter 4, we found that newly-developed malicious apps are recognized as malicious by a noticeably smaller subset of VirusTotal scanners. Hence, VirusTotal's dynamicity has more impact on their scan reports than on the scan reports of older apps. So, we may answer this research question as follows:

We speculate that there are many properties of an Android app that may prevent VirusTotal from correctly labeling it. We identified some of those properties to be (1) the malware type to which a malicious app belongs, (2) the source of the (malicious) apps (i.e., found on a marketplace versus in-house developed), (3) the frequency of encountering the app on user devices (i.e., its popularity), and (4) the age of the app. Malicious apps that belong to malware types whose malignancy is debatable (e.g., Adware), are in-house developed and not uploaded to app marketplaces, and are newly developed are unlikely to be detected by the scanners used by VirusTotal.

RQ3: Is there a universal ensemble of VirusTotal scanners that are more correct over time than others?

As an alternative to threshold-based labeling strategies that rely on the verdicts of random VirusTotal scanners, the research community attempted to find a set of VirusTotal scanners whose verdicts are more reliable. Arp et al., for instance, devised a threshold-based labeling strategy that focuses on the verdicts of ten VirusTotal scanners they believed to be reliable [15]. In Section 5.2, we used an extended version of Mohaisen et al.'s definition of correct VirusTotal scanners in [92] to find out whether there is a recurring or universal set of VirusTotal scanners that correctly label apps as malicious and benign regardless of the time period. We used the scan reports of apps in the *AMD+GPlay*, *Hand-Labeled*, and *Hand-Labeled 2019* datasets gathered between July 5^{th}, 2019 and November 8^{th}, 2019 to calculate the set of correct VirusTotal scanners based on the ground truth that we possessed for those dataset (see Section 1.3.4 for details on how we obtained such a ground truth). We found that the set of correct VirusTotal scanners differs from one dataset to another, both in terms of members and cardinality. Furthermore, we compared the set correct VirusTotal scanners calculated based on the scan reports of apps in the *AMD+GPlay* dataset gathered between November 2018 and November 8^{th}, 2019, and found that

the two sets of scanners differ as well. So, we may answer this research question as follows:

Our measurements show that the set of correct `VirusTotal` scanners changes depending on the composition of the dataset whose `VirusTotal` scan reports are used to find those scanners and on the time period within which those scan reports were gathered. Consequently, there is no point in pursuing a universal set of `VirusTotal` scanners that would always return correct labels across datasets and periods of time.

RQ4: How can we standardize the interpretation of `VirusTotal` scan reports to produce accurate labels for Android apps?

The research community devises ad hoc, subjective labeling strategies to label Android apps based on their `VirusTotal` scan reports because there are no standard methods to interpret the information in these reports. As discussed throughout this thesis, the dynamicity of `VirusTotal` undermines the accuracy of labels given by these strategies. In this context, we attempted to provide the research community with labeling strategies that bypass or withstand `VirusTotal`'s dynamicity to provide labels that better reflect the malignancy of apps, effectively standardizing the process of utilizing `VirusTotal` scan reports to label apps. In Chapter 4, we identified one aspect of `VirusTotal`'s dynamicity that undermines the labeling accuracy of threshold-based labeling strategies using a fixed threshold for prolonged periods. We used this finding to devise an algorithm that finds the currently optimal threshold of `VirusTotal` scanners to use in labeling apps as malicious and benign at any point in time. The usefulness of the algorithm was constrained by a number of conditions that are not always feasible. In Chapter 5, we implemented a framework, Maat, that automates the process of analyzing `VirusTotal` scan reports and devising labeling strategies. With little to no user intervention, Maat trains ML-based labeling strategies that, once trained, can maintain stable labeling accuracies for periods of time between seven and 12 months without needing to be retrained (see Chapter 6). This is made possible by the fact that Maat's ML-based labeling strategies rely on random forests of 100 decision trees that use the verdicts of three to four `VirusTotal` scanners. So, we may answer this research question as follows:

Standardizing the interpretation of `VirusTotal`'s scan reports can be achieved by relieving researchers of the burden of having to devise subjective thresholds of `VirusTotal` scanners to discern the malignancy of an app based on its scan report. In this thesis, we devised two methods that can standardize the utilization of `VirusTotal`. We propose an algorithm that finds the currently optimal thresholds of `VirusTotal` scanners to use to label apps. We also developed a framework that automates the process of analyzing `VirusTotal`

scan reports to build ML-based labeling strategies that are more resilient to `VirusTotal`'s dynamicity than conventional threshold-based labeling strategies.

RQ5: What are the aspects of `VirusTotal`'s dynamicity that impact the performance of labeling strategies, particularly threshold-based ones?

In Chapter 4, we found that `VirusTotal` frequently changes the set of scanners included in the scan reports of apps, regardless of whether those scanners correctly labeled such apps. This change usually alters the number of `VirusTotal` scanners deeming an app as malicious in an app's scan report, which is the value that threshold-based labeling strategies rely on to label apps as malicious and benign. We noticed, in Section 4.2, the impact of this change could be noticed upon examining the labeling accuracy of threshold-based labeling strategies that rely on fixed thresholds against newly-developed malicious apps in the *Hand-Labeled 2019* dataset, especially since the number of `VirusTotal` scanners deeming them as malicious is already low courtesy of the apps' novelty. Furthermore, in Chapter 5, we found that the platform unexpectedly switches the versions of `VirusTotal` scanners it uses to label apps to ones that may not be adequate to analyze and detect malicious Android apps. This causes the performance of scanners that are known to be competent within the market, such as `BitDefender`, to deteriorate significantly. However, unlike changing the set of scanners across scan dates, changing the versions of scanners appears to be less frequent. These manipulations of scanners in the `VirusTotal` scan reports of apps had an impact on both the structure and labeling accuracy of Maat's ML-based labeling strategies. As seen in Chapter 6, removing some scanners from the `VirusTotal` scan report of an app causes the mislabeling of a few apps during the training phase of ML-based labeling strategies, which causes the validation scores of each iteration during grid search to slightly differ. These insignificant differences might encourage Maat to choose hyperparameters for the ML-based labeling strategies that yield shallow random forests that are incapable of correctly classifying apps, especially newer ones in the *Hand-Labeled 2019* dataset. As for the labeling accuracy of Maat's ML-based labeling strategies, they seemed to suffer from the same problem of removing a set of `VirusTotal` scanners from the scan reports of malicious apps in the *Hand-Labeled 2019* dataset. So, we may answer this research question as follows:

Among the considered aspects for `VirusTotal`'s dynamicity, we identified four aspects that impact the performance of threshold-based and ML-based labeling strategies are (a) the frequent and unexpected manipulation of the sets and verdicts of scanners included by `VirusTotal` in the scan reports of apps, and (b) the change of the versions of `VirusTotal` scanners with ones that may not be adequate to scan and detect Android malicious apps. While such frequent manipulation has a small impact on the `VirusTotal` scan reports

of older Android apps (i.e., due to their relative maturity and stability), it noticeably impacts the performance of such labeling strategies upon labeling newly-developed Android apps with less mature `VirusTotal` scan reports.

RQ6: What are the limitations of `VirusTotal` and how can they be mitigated?

Despite calls within the research community to replace `VirusTotal` with a more reliable alternative, the online platform continues to be utilized by researchers to label apps in the datasets they use to evaluate their malware detection methods. These calls for replacement stem from the common knowledge that platforms, such as `VirusTotal`, suffer from some drawbacks. However, these drawbacks and their impact on the process of labeling apps were neither thoroughly discussed nor demonstrated. Throughout this doctoral thesis, we discussed some of those limitations and their impacts on identifying the set of correct scanners, estimating the time it takes scan reports to stabilize, and the performance of different labeling strategies. Using the insights we gained from our measurements and experiments, in Chapter 7, we succinctly enumerate four limitations that we categorized into (a) the dynamicity of `VirusTotal`'s scan reports, and (b) the lack of access to re-scanned scan reports. To mitigate those limitations, we proposed the architecture of an alternative online platform, Eleda, that can be used to label (Android) apps. The proposed platform is designed to maintain the same set of scanners to scan and label apps, to use the versions of scanners that are implemented to detect Android apps, to frequently re-scan apps using the same set of scanners to enable studying the performance of scanners over time, and to grant free access to those re-scan results. So, we may answer this research question as follows:

We identified four main limitations to `VirusTotal` that jeopardize its usefulness. Firstly, the platform uses scanners or versions of scanners that are not suitable to detect Android malware. Secondly, for reasons unknown to us, the platform changes the set of scanners it uses to scan the same apps over time, which undermines the sustainability of labeling strategies, such as threshold-based ones. Thirdly, the platform does not automatically re-scan apps and relies on manually re-scanning apps either via its web-interface or via remote API requests. Lastly, the platform does not grant access to the history of scans, effectively preventing researchers from studying the performance of scanners over extended periods of time.

9.2. Literature Gaps and Contributions

In Section 1.2, we discussed a number of literature gaps associated with utilizing `VirusTotal` scan reports to label Android apps as malicious and benign. Those gaps can be summarized as follows:

1. The scan results in a VirusTotal scan report are treated as ground truth and, hence, pose an upper bound on the performance of labeling strategies based on those scan results (i.e., no labeling strategy can perform better than the ground truth). However, such scan results do not always provide correct verdicts vis-à-vis the type of an app being malicious or benign. To the best of our knowledge, there are no efforts in the literature that attempt to find the characteristics of an app or under which circumstances a VirusTotal scan report would fail to provide the platform's users with accurate verdicts that reflect an app's ground truth.

2. VirusTotal acts as a meta-scanner that provides the scan results of different antiviral scanners; it delegates the task of deeming an app as malicious or benign based on its scan report to the users. Unfortunately, there are no standards in the literature on how to use the aforementioned scan results to label Android apps. This forces researchers to devise their own ad-hoc strategies to label Android apps, which usually rely on a fixed threshold of positives (i.e., malicious label), to deem an app as malicious.

3. It is common knowledge within the research community that VirusTotal is a dynamic platform that frequently changes. However, researchers continue to rely on the platform's scan reports to label the apps they use to evaluate, for example, their newly-devised malware detection methods. With no clear enumeration of the aspects of VirusTotal's dynamicity and how they impact the labeling accuracies of different types of labeling strategies, researchers risk adopting labeling strategies that yield incorrect labels and, in turn, provide researchers with false assessments of their malware detection methods.

In addressing those literature gaps, we developed algorithms and platforms and conducted measurements and experiments that make the following contributions:

1. Given that the quality of labels given by any labeling strategy is based on VirusTotal's scan reports, we expected to find methods in the literature that instruct researchers on how to avoid unstable VirusTotal scan reports. However, to the best of our knowledge, there are no efforts that address what we defined as the first literature gap. In Chapter 2 and Chapter 5, we reveal some of the properties that can make it difficult for VirusTotal scanners to detect the malignancy of a malicious Android app, which undermines the stability and reliability of this app's VirusTotal scan reports. In particular, we found that some malware types, such as Adware, the age of an app and its VirusTotal scan report, and whether the app can be found in a marketplace are indications of whether VirusTotal scanners can detect the malignancy of an app. Furthermore, to contribute to addressing this gap, we enumerate the four limitations we found VirusTotal to be suffering from and, in Chapter 7, propose a design of a more stable and reliable alternative to the platform.

2. The main contribution of this thesis dwells in its effort to standardize the process of using VirusTotal scan reports of Android apps to label them as malicious

and benign, which addresses the second literature gap of lacking such standards to interpret VirusTotal scan reports to label apps as such. In Chapter 4, we propose a generic method to devise threshold-based labeling strategies that use the currently optimal number of VirusTotal scanners needed to label an app as malicious, instead of using a fixed threshold as common within the research community. Furthermore, we introduced in Chapter 5 a framework we implemented to automate the process of analyzing VirusTotal scan reports to devise ML-based labeling strategies that need not be frequently retrained yet is more resilient to VirusTotal's dynamicity. Our experiments in Chapter 6 indicate that Maat's ML-based labeling strategies can match the labeling accuracy of the best possible thresholds at any point in time and can contribute to training more effective ML-based detection methods.

3. The research community claims that VirusTotal's dynamicity is common knowledge. Yet, researchers continue to make the same mistake of utilizing threshold-based labeling strategies that rely on fixed thresholds for prolonged periods to label apps they use to evaluate newly-devised ML-based labeling strategies. To the best of our knowledge, there is little to know work that explicitly points out the aspects of VirusTotal's dynamicity, how it manifests, and how it impacts threshold-based labeling strategies, which defines the third literature gap. To address this gap, in Chapter 4 and Chapter 5, we reveal the aspects of VirusTotal's dynamicity and how they impacted some of the renowned threshold-based labeling strategies and Maat's ML-based labeling strategies.

9.3. Limitations

VirusTotal-Based Ground Truth as an Upper Bound. In this thesis, we focus on VirusTotal as the source of scanner verdicts upon which labeling strategies are based. As discussed in Section 1.2, this puts an upper bound on the labeling accuracies that can be achieved by **any** labeling strategy upon labeling apps based on their VirusTotal scan reports. One can notice, for example, that the MCC scores achieved by the best performing labeling strategies in Section 6.1 never reached 1.0 (i.e., perfectly accurate labeling) on either the *Hand-Labeled* or *Hand-Labeled 2019* datasets. This behavior is a result of the VirusTotal scanners not deeming apps that we manually deemed malicious (e.g., because they track the GPS location of users without the users' knowledge), as such. Consequently, all labeling strategies, including Maat's ML-based labeling strategies and the Best Thresholds, mislabeled those apps as benign because they base their labels on the verdicts of VirusTotal scanners. However, this limitation does not undermine the contributions of this thesis for the following reasons. The infeasibility of manually labeling Android apps as malicious and benign forces researchers to use online platforms, such as VirusTotal, as the source of ground truth for the apps they use to train and evaluate their detection methods. We assume that the continuity of using VirusTotal implies that researchers accept this limitation of relying on machine-generated ground truth. Based on

this assumption, in this thesis, we attempt to provide the research community with insights, algorithms, and frameworks that optimally utilize VirusTotal to label Android apps. Firstly, we identify a number of properties that might contribute to an app, malicious or benign, being mislabeled; these cues are meant to help researchers avoid VirusTotal scan reports that are unreliable. Secondly, we devised an algorithm to devise threshold-based labeling strategies and implemented a framework, Maat, to automatically train ML-based labeling strategies that are meant to be less susceptible to VirusTotal's dynamicity and limitations and, thus, provide the research community with the best possible outcome from using VirusTotal. To that end, thirdly, we enumerate the aspects of VirusTotal's dynamicity and the resulting limitations and provide the community with a blueprint of an alternative to VirusTotal that mitigates those limitations.

Partial Reliance on VirusTotal's Ground Truth. To label apps in the *AMD+GPlay* dataset, we relied on the labels generated by Wei et al. to label apps in the *AMD* dataset, which combined filtration of malicious apps using the vt$\geq$50% labeling strategy and manual analysis to accurately label apps in the dataset as malicious. We also used the VirusTotal scan reports of apps in the *GPlay* dataset, which were downloaded from the well-vetted Google Play store, to deem them as benign according to the criterion *positives==0* between November 2018 and November 8^{th}, 2019. The threat to validity, in this case, is whether those measures we took to ensure the accuracy of the labels in this dataset were not enough. Since we rely on the *AMD+GPlay* dataset to train Maat's ML-based labeling strategies, inaccuracies in the labels of those apps threaten to undermine the credibility of our findings. To ensure the credibility of our results, we plan on manually-labeling as many Android apps as possible, use them for training Maat's ML-based labeling strategies, and comparing the results we achieve with the results in this thesis. The main obstacle, in this case, is that manually analyzing thousands of apps is a lengthy process.

Confinement to Used Datasets and Scan Reports. The results we recorded and discussed in this thesis are confined to the datasets that we used and the period within which the VirusTotal scan reports were gathered. So, the threat to validity, in this case, is whether the same conclusions we drew can be reached upon using other datasets. For example, would Maat's ML-based labeling strategies always manage to mimic the performance of their threshold-based counterparts in terms of accurately labeling apps? In Chapter 6, in addition to discussing that the structure of Maat's ML-based labeling strategies are less susceptible to changes in VirusTotal's scan reports, we attempted to simulate this process examining the performance of ML-based and threshold-based labeling strategies over a period of time. However, the experiments were conducted on the same datasets, and the test time period spanned a period of four months (i.e., between July 5^{th}, 2019, and November 8^{th}, 2019). As discussed earlier, access to old VirusTotal scan reports is only available under commercial licenses, which we consider a limitation of VirusTotal from an academic perspective. Furthermore, beyond November 8^{th}, 2019, we found the

academic license we possess, which enabled us to re-scan apps in our datasets and to download their up-to-date scan reports, was not granted the re-scan permission anymore. Effectively, comparing the performance of Maat's ML-based labeling strategies against threshold-based ones in the future seems to be hindered. To address the first limitation, we plan to use other datasets to run our experiments. This can also be simulated by using random subsets of the *AMD+GPlay* dataset to train Maat's ML-based labeling strategies. As for the second limitation, we argue that the adequate method to address this limitation is to find an alternative to `VirusTotal`, which includes implementing a platform that mitigates `VirusTotal`'s limitations that we identified in this thesis.

Small Size of Test Datasets. We base a number of our insights on results achieved against two small datasets, namely *Hand-Labeled* and *Hand-Labeled 2019*. In particular, these two datasets are used to demonstrate the dynamicity of `VirusTotal` in Section 4.2, compare the labeling accuracy of different labeling strategies in Section 6.1, and evaluate the detection capabilities of ML-based detection methods labeled using different labeling strategies in Section 6.4. Needless to say, the larger the sizes of these datasets, the more reliable the results based on them will be. In other words, the threat to validity, in this case, is that our results may not be reliable. The reason behind the relatively small size of the datasets is that we had to manually analyze their apps to ensure accurate ground truth (i.e., malicious and benign), especially since we those apps to evaluate the accuracy of labeling strategies based on `VirusTotal` scan reports. Manually labeling thousands of apps is an infeasible process, as discussed in Chapter 1. To compensate for the size of the datasets, we acquired a random sample of apps from *AndroZoo* that span different ages, sources (i.e., marketplaces), app categories (e.g., games versus utilities), sizes, etc. However, aware of the possible limitation, we confine the results of our experiments and measurements to the utilized datasets and aspire to further reinforce them with future work.

Generalization to other detection methods. To further motivate the significance of our work, we showed that accurate labeling generally enhances the performance of ML-based detection methods trained using static features. This begs the question of whether we can replicate the same results if we utilize different features and/or different classifiers. The most appropriate method to verify this hypothesis is to conduct the same experiments using different types of features and classifiers in pursuit of counter-examples. However, there is a plethora of work in this area that spans a multitude of approaches to malware analysis and detection [142], which indeed cannot be comprehensively covered in this work. So, we plan to conduct more experiments using different types of features (e.g., dynamic features), and/or classifiers to further solidify our findings in Chapter 4 and Chapter 6.

9.4. Future Work

In this section, we enumerate the future work that can either address the limitations we discussed in the previous section or build on the work presented in this doctoral thesis towards developing more effective malware detection methods.

Application to Other Domains. We focused on Android apps and their `VirusTotal` scan reports. However, having found that ML-based labeling strategies perform better using naive features, we are positive that the Maat method can seamlessly generalize to other domains (e.g., Windows-based malware), especially since `VirusTotal` scan reports always contain the verdicts of different antiviral scanners in addition to domain-specific information. In the future, we can imagine performing the measurements and experiments we conducted in this thesis on scan reports of apps belonging to other domains. Another aspect of generalization for Maat that we plan to explore is the utilization of more granular labels. That is, instead of using binary labels of malicious versus benign, we plan to use labels that, for example, depict an app's malware type or family (e.g., `Adware`, `Ransom`, `Dowgin`, etc.).

Application to Other Detection Methods. In this thesis, we focused on static ML-based detection methods and showed that accurate labeling, especially using Maat's ML-based strategies, enhances the effectiveness of such detection methods. To test whether this positive impact on detection generalizes to other detection methods, we plan to use different types of detection methods and measure whether their detection capabilities of (Android) malware increases after being trained using apps labeled with Maat's ML-based strategies.

Alternative to `VirusTotal`. We demonstrated four limitations to `VirusTotal` in this thesis, the most important of which the platform's utilization of older, inadequate versions of scanners and its frequent change of the set of scanners it uses to scan apps. To mitigate these limitations, we proposed the development of an academic, non-profit alternative platform, Eleda. In the future, we aspire to contribute to the development and evaluation of such a platform that uses the same versions of scanners that can be found in the market (e.g., in app marketplaces), and offers the information it gathers about different apps to malware analysis and detection the research community.

History of Scan Reports. Given the infeasibility of access earlier scan reports for the datasets we used in this thesis, we plan on building a history of scan reports for apps recently-crawled by platforms, such as *AndroZoo*, by re-scanning these apps every two weeks and downloading their scan reports for an extended period, either using `VirusTotal` or Eleda. Using such reports, we can study the evolution of scan reports, including the performance of different scanners. Furthermore, we can conduct experiments that

investigate whether the number of scanners deeming an app as malicious converges to a specific range after some time, effectively testing other definitions of stability.

Bibliography

[1] Packadroid – a framework for repackaging android applications, 2018.

[2] Get paid to show relevant ads from over a million advertisers with google admob, 2019.

[3] K-9 mail - advanced email for android, 2019.

[4] Yousra Aafer, Wenliang Du, and Heng Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *International Conference on Security and Privacy in Communication Systems*, pages 86–103. Springer, 2013.

[5] Leonard M Adleman. An abstract theory of computer viruses. In *Conference on the Theory and Application of Cryptography*, pages 354–374. Springer, 1988.

[6] admin Comodo Antivirus. What is reddrop malware? — its impact on android devices, 2019.

[7] Mansour Ahmadi, Angelo Sotgiu, and Giorgio Giacinto. Intelliav: Toward the feasibility of building intelligent anti-malware on android devices. In *International Cross-Domain Conference for Machine Learning and Knowledge Extraction*, pages 137–154. Springer, 2017.

[8] Jona Neumeier Aleieldin Salem, Michael Hesse and Alexander Pretschner. Towards empirically assessing behavior stimulation approaches for android malware. In *The 13th International Conference on Emerging Security Information, Systems and Technologies*. International Academy, Research and Industry Association (IARIA), 2019.

[9] Sebastian Banescu Aleieldin Salem and Alexander Pretschner. Maat: Automatically analyzing virustotal for accurate labeling and effective malware detection. 2020.

[10] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*, pages 468–471. IEEE, 2016.

[11] Kevin Allix, Quentin Jerome, Tegawende 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*, pages 384–393. IEEE, 2014.

[12] Hybrid Analysis. Free automated malware analysis service, 2019.

[13] androguard. androguard: Reverse engineering, malware and goodware analysis of android applications ... and more (ninja !), 2018.

[14] Apktool. Apktool, 2018.

[15] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*, 2014.

[16] Saba Arshad, Munam Ali Shah, Abid Khan, and Mansoor Ahmed. Android malware detection & protection: a survey. *International Journal of Advanced Computer Science and Applications*, 7:463–475, 2016.

[17] 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 *Acm Sigplan Notices*, volume 49, pages 259–269. ACM, 2014.

[18] Sebastian Banescu, Tobias Wüchner, Aleieldin Salem, Marius Guggenmos, Martin Ochoa, and Alexander Pretschner. A Framework for Empirical Evaluation of Malware Detection Resilience Against Behavior Obfuscation. In *10th International Conference on Malicious and Unwanted Software*, Fajardo, Puerto Rico, 2015.

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

[20] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. Statistical deobfuscation of android applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 343–355, 2016.

[21] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[22] Bitdefender. Bitdefender cements spot as world av industry leader in 2011, 2011.

[23] Aaron Brown. This is how much money google really makes from android, 2016.

[24] John Charlton, Pang Du, Jin-Hee Cho, and Shouhuai Xu. Measuring relative accuracy of malware detectors in the absence of ground truth. In *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*, pages 450–455. IEEE, 2018.

[25] Thomas M Chen and Jean-Marc Robert. The evolution of viruses and worms. *Statistical methods in computer security*, 1, 2004.

[26] Catalin Cimpanu. Almost 150 million users impacted by new simbad android adware, 2019.

[27] OPSWAT Metadefender Cloud. Opswat metadefender cloud — multiscanning, deep cdr, and sandbox api, 2019.

[28] Fred Cohen. *Computer viruses*. PhD thesis, University of Southern California Doctoral dissertation, 1986.

[29] Fred Cohen. Computer viruses: theory and experiments. *Computers & security*, 6(1):22–35, 1987.

[30] Jedidiah R Crandall, Gary Wassermann, Daniela AS de Oliveira, Zhendong Su, S Felix Wu, and Frederic T Chong. Temporal search: Detecting hidden malware timebombs with virtual machines. In *ACM SIGARCH Computer Architecture News*, volume 34, pages 25–36. ACM, 2006.

[31] CrowdStrike. Crowdstrike falcon antivirus replacement, 2020.

[32] 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. *IEEE Transactions on Dependable and Secure Computing*, 2017.

[33] Android Developers. Activity, 2019.

[34] Android Developers. Build your aoo from the command line, 2019.

[35] Android Developers. Save data using sqlite, 2019.

[36] Android Developers. Shrink, obfuscate, and optimize your app, 2019.

[37] Android Developers. Timer, 2019.

[38] 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*, pages 172–192. Springer, 2018.

[39] Joshua J Drake, Zach Lanier, Collin Mulliner, Pau Oliva Fora, Stephen A Ridley, and Georg Wicherski. *Android hacker's handbook*. John Wiley & Sons, 2014.

[40] Pang Du, Zheyuan Sun, Huashan Chen, Jin-Hee Cho, and Shouhuai Xu. Statistical estimation of malware detection metrics in the absence of ground truth. *IEEE Transactions on Information Forensics and Security*, 13(12):2965–2980, 2018.

[41] Sumeet Dua and Xian Du. *Data mining and machine learning in cybersecurity*. CRC press, 2011.

[42] Ken Dunham, Shane Hartman, Manu Quintans, Jose Andre Morales, and Tim Strazzere. *Android malware and analysis*. Auerbach Publications, 2014.

[43] Karim O Elish, Xiaokui Shu, Danfeng Daphne Yao, Barbara G Ryder, and Xuxian Jiang. Profiling user-trigger dependence for android malware detection. *Computers & Security*, 49:255–273, 2015.

[44] Roberto Jordaney Johannes Kinder Feargus Pendlebury, Fabio Pierazzi and Lorenzo Cavallaro. Tesseract: Eliminating experimental bias in malware classification across space and time. In *28th USENIX Security Symposium*, Santa Clara, CA, 2019. USENIX Association.

[45] Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Ainuddin Wahid Abdul Wahab. A review on feature selection in mobile malware detection. *Digital investigation*, 13:22–37, 2015.

[46] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Triggerscope: Towards detecting logic bombs in android applications. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 377–396. IEEE, 2016.

[47] 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*, pages 45–54. ACM, 2013.

[48] GReaT. Viceleaker operation: mobile espionage targeting middle east, 2019.

[49] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial examples for malware detection. In *European Symposium on Research in Computer Security*, pages 62–79. Springer, 2017.

[50] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *The Journal of Machine Learning Research*, 3:1157–1182, 2003.

[51] Isabelle Guyon and André Elisseeff. An introduction to feature extraction. In *Feature extraction*, pages 1–25. Springer, 2006.

[52] Hara Hiroaki, Lilang Wu, and Lorin Wu. New version of xloader that disguises as android apps and an ios profile holds new links to fakespy, 2019.

[53] Johannes Hoffmann, Teemu Rytilahti, Davide Maiorca, Marcel Winandy, Giorgio Giacinto, and Thorsten Holz. Evaluating analysis tools for android apps: Status quo and robustness against obfuscation. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 139–141. ACM, 2016.

[54] Médéric Hurier. *Creating better ground truth to further understand Android malware: A large scale mining approach based on antivirus labels and malicious artifacts*. PhD thesis, University of Luxembourg, 2019.

[55] Médéric Hurier, Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. On the lack of consensus in anti-virus decisions: Metrics and insights on building ground truths of android malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 142–162. Springer, 2016.

[56] Médéric Hurier, Guillermo Suarez-Tangil, Santanu Kumar Dash, Tegawendé F Bissyandé, Yves Le Traon, Jacques Klein, and Lorenzo Cavallaro. Euphony: Harmonious unification of cacophonous anti-virus vendor labels for android malware. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 425–435. IEEE Press, 2017.

[57] Nwokedi Idika and Aditya P Mathur. A survey of malware detection techniques. *Purdue University*, 48:2007–2, 2007.

[58] Muhammad Ikram, Narseo Vallina-Rodriguez, Suranga Seneviratne, Mohamed Ali Kaafar, and Vern Paxson. An analysis of the privacy and security risks of android vpn permission-enabled apps. In *Proceedings of the 2016 Internet Measurement Conference*, pages 349–364, 2016.

[59] AV-Test: The Independent IT-Security Institute. The best antivirus software for android, 2019.

[60] MIT App Inventor. About us - explore mit app inventor, 2019.

[61] Richard Jensen and Qiang Shen. *Computational intelligence and feature selection: rough and fuzzy approaches*, volume 8. John Wiley & Sons, 2008.

[62] Alex Kantchelian, Sadia Afroz, Ling Huang, Aylin Caliskan Islam, Brad Miller, Michael Carl Tschantz, Rachel Greenstadt, Anthony D Joseph, and JD Tygar. Approaches to adversarial drift. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 99–110. ACM, 2013.

[63] Alex Kantchelian, Michael Carl Tschantz, Sadia Afroz, Brad Miller, Vaishaal Shankar, Rekha Bachwani, Anthony D Joseph, and J Doug Tygar. Better malware ground truth: Techniques for weighting anti-virus vendor labels. In *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security*, pages 45–56. ACM, 2015.

[64] Kaspersky. Machine learning methods for malware detection, 2020.

[65] Stefan Katzenbeisser, Johannes Kinder, and Helmut Veith. *Malware Detection*, pages 752–755. Springer US, 2011.

[66] Tom Kelchner. The (in) consistent naming of malcode. *Computer Fraud & Security*, 2010(2):5–7, 2010.

[67] Joxean Koret and Elias Bachaalany. *The Antivirus Hacker's Handbook*. Wiley Online Library, 2015.

[68] Simon Kramer and Julian C Bradfield. A general definition of malware. *Journal in computer virology*, 6(2):105–114, 2010.

[69] Satheesh kumar Sasidharan and Ciza Thomas. A survey on metamorphic malware detection based on hidden markov model. In *2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 357–362. IEEE, 2018.

[70] Rita Lao. China says apps should get user consent before tracking, 2019.

[71] Riccardo Leardi, R Boggia, and M Terrile. Genetic algorithms as a strategy for feature selection. *Journal of chemometrics*, 6(5):267–281, 1992.

[72] Charles LeDoux and Arun Lakhotia. Malware and machine learning. In *Intelligent Methods for Cyber Warfare*, pages 1–42. Springer, 2015.

[73] Lexico. malware — definition of malware in english by lexico dictionaries, 2019.

[74] Li Li, Tegawendé Bissyandé, and Jacques Klein. Rebooting research on detecting repackaged android apps: Literature review and benchmark. *arXiv preprint arXiv:1811.08520*, 2018.

[75] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. Understanding android app piggybacking: A systematic study of malicious code grafting. *IEEE Transactions on Information Forensics and Security*, 12(6):1269–1284, 2017.

[76] Li Li, Daoyuan Li, Tegawendé François D Assise Bissyande, Jacques Klein, Haipeng Cai, David Lo, and Yves Le Traon. Automatically locating malicious packages in piggybacked android apps. In *4th IEEE/ACM International Conference on Mobile Software Engineering and Systems*, 2017.

[77] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Droidbot: a lightweight ui-guided test input generator for android. In *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*, pages 23–26. IEEE, 2017.

[78] 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*, pages 192–214. Springer, 2017.

[79] Ying-Dar Lin, Yuan-Cheng Lai, Chien-Hung Chen, and Hao-Chuan Tsai. Identifying android malicious repackaged applications by thread-grained system call sequences. *computers & security*, 39:340–350, 2013.

[80] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. Andrubis-1,000,000 apps later: A view on current android malware behaviors. In *Proceedings of the the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

[81] Bing Liu, Wynne Hsu, Yiming Ma, et al. Integrating classification and association rule mining. In *KDD*, volume 98, pages 80–86, 1998.

[82] Symphony Luo and Peter Yan. Fake apps: Feigning legitimacy, 2014.

[83] PC Magazin. Bitdefender free edition, 2008.

[84] Federico Maggi, Andrea Bellini, Guido Salvaneschi, and Stefano Zanero. Finding non-trivial malware naming inconsistencies. In *International Conference on Information Systems Security*, pages 144–159. Springer, 2011.

[85] Malwarebytes. Malware, 2019.

[86] Malwr. Coming back soon!, 2019.

[87] 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 of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2017.

[88] Joseph Menn. Exclusive: Russian antivirus firm faked malware to harm rivals - ex-employees, 2015.

[89] Abraham H Mhaidli, Yixin Zou, and Florian Schaub. " we can't live without them!" app developers' adoption of ad networks and their considerations of consumer risks. In *Fifteenth Symposium on Usable Privacy and Security ({SOUPS} 2019)*, 2019.

[90] Microsoft. Defining malware: Faq, 2019.

[91] Brad Miller, Alex Kantchelian, Michael Carl Tschantz, Sadia Afroz, Rekha Bachwani, Riyaz Faizullabhoy, Ling Huang, Vaishaal Shankar, Tony Wu, George Yiu, et al. Reviewer integration and performance measurement for malware detection. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 122–141. Springer, 2016.

[92] Aziz Mohaisen and Omar Alrawi. Av-meter: An evaluation of antivirus scans and labels. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 112–131. Springer, 2014.

[93] Aziz Mohaisen, Omar Alrawi, Matt Larson, and Danny McPherson. Towards a methodical evaluation of antivirus scans and labels. In *International Workshop on Information Security Applications*, pages 231–241. Springer, 2013.

[94] Luis Carlos Molina, Lluís Belanche, and Àngela Nebot. Feature selection algorithms: A survey and experimental evaluation. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*, pages 306–313. IEEE, 2002.

[95] Marcin Moskala and Igor Wojda. *Android Development with Kotlin*. Packt Publishing Ltd, 2017.

[96] Kevin Murnane. How to protect yourself from the global wanacry ransomware attack, 2017.

[97] Fairuz Amalina Narudin, Ali Feizollah, Nor Badrul Anuar, and Abdullah Gani. Evaluation of machine learning classifiers for mobile malware detection. *Soft Computing*, 20(1):343–357, 2016.

[98] Jon Oberheide and Charlie Miller. Dissecting the android bouncer. *SummerCon2012, New York*, 2012.

[99] Tavis Ormandy. Sophail: A critical analysis of sophos antivirus. *Proc. of Black Hat USA*, 2011.

[100] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. Enabling fair ml evaluations for security. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2264–2266. ACM, 2018.

[101] Peng Peng, Limin Yang, Linhai Song, and Gang Wang. Opening the blackbox of virustotal: Analyzing online phishing scan engines. In *Proceedings of the Internet Measurement Conference*, pages 478–485. ACM, 2019.

[102] Roberto Perdisci et al. Vamo: towards a fully automated malware clustering validity analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 329–338. ACM, 2012.

[103] Gavin Phillips. These 4 antivirus tools are using ai to protect your system, 2018.

[104] Paul Prasse, Lukáš Machlica, Tomáš Pevnỳ, Jiří Havelka, and Tobias Scheffer. Malware detection by analysing encrypted network traffic with neural networks. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 73–88. Springer, 2017.

[105] Radware. The history of malware, 2015.

[106] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. In *NDSS*, 2016.

[107] Siegfried Rasthofer, Steven Arzt, Stefan Triller, and Michael Pradel. Making malory behave maliciously: Targeted fuzzing of android execution environments. In *Proceedings of the 39th International Conference on Software Engineering*, pages 300–311. IEEE Press, 2017.

[108] 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*, pages 187–202. Springer, 2015.

[109] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[110] Markus Ringnér. What is principal component analysis? *Nature biotechnology*, 26(3):303–304, 2008.

[111] Simon Rogers and Mark Girolami. *A first course in machine learning*. CRC Press, 2011.

[112] Neil J. Rubenking. False positives sink antivirus ratings, 2015.

[113] Shefali Sachdeva, Romuald Jolivot, and Worawat Choensawat. Android malware classification based on mobile security framework. *IAENG International Journal of Computer Science*, 45(4), 2018.

[114] Muhammad N Sakib, Chin-Tser Huang, and Ying-Dar Lin. Maximizing accuracy in multi-scanner malware detection systems. *Computer Networks*, 169:107027, 2020.

[115] Aleieldin Salem. Stimulation and detection of android repackaged malware with active learning. *arXiv preprint arXiv:1808.01186*, 2018.

[116] Aleieldin Salem. Towards accurate labeling of android apps for reliable malware detection. In *The 31st International Symposium on Software Reliability Engineering (ISSRE) [Under Review]*. arXiv, 2020.

[117] Aleieldin Salem and Sebastian Banescu. Metadata recovery from obfuscated programs using machine learning. In *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*, page 1. ACM, 2016.

[118] Aleieldin Salem, F. Franziska Paulus, and Alexander Pretschner. Repackman: A tool for automatic repackaging of android apps. In *Proceedings of the 1st International Workshop on Advances in Mobile App Analysis*, pages 25–28. ACM, 2018.

[119] Aleieldin Salem and Alexander Pretschner. Poking the bear: Lessons learned from probing three android malware datasets. In *Proceedings of the 1st International Workshop on Advances in Mobile App Analysis*, pages 19–24. ACM, 2018.

[120] Aleieldin Salem, Tabea Schmidt, and Alexander Pretschner. Idea: Automatic localization of malicious behaviors in android malware with hidden markov models. In *International Symposium on Engineering Secure Software and Systems*, pages 108–115. Springer, 2018.

[121] Hillary Sanders and Joshua Saxe. Garbage in, garbage out: how purportedly great ml models can be screwed up by bad data. *Technical report*, 2017.

[122] Borja Sanz, Igor Santos, Carlos Laorden, Xabier Ugarte-Pedrero, and Pablo Garcia Bringas. On the automatic categorisation of android applications. In *2012 IEEE Consumer communications and networking conference (CCNC)*, pages 149–153. IEEE, 2012.

[123] Borja Sanz, Igor Santos, Carlos Laorden, Xabier Ugarte-Pedrero, Pablo Garcia Bringas, and Gonzalo Álvarez. Puma: Permission usage to detect malware in android. In *International Joint Conference CISIS'12-ICEUTE´ 12-SOCO´ 12 Special Sessions*, pages 289–298. Springer, 2013.

[124] Ryo Sato, Daiki Chiba, and Shigeki Goto. Detecting android malware by analyzing manifest files. *Proceedings of the Asia-Pacific Advanced Network*, 36:23–31, 2013.

[125] scikit learn. Classifier comparison, 2019.

[126] scikit learn. Feature selection, 2019.

[127] scikit learn. Gridsearchcv, 2019.

[128] scikit learn. Randomsearchcv, 2019.

[129] scikit learn. sklearn.metrics.matthews_corrcoef, 2019.

[130] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. Avclass: A tool for massive malware labeling. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 230–253. Springer, 2016.

[131] Hossain Shahriar and Victor Clincy. Kullback-leibler divergence based detection of repackaged android malware. 2015.

[132] Yuru Shao, Xiapu Luo, Chenxiong Qian, Pengfei Zhu, and Lei Zhang. Towards a scalable resource-driven approach for detecting repackaged android applications. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 56–65. ACM, 2014.

[133] Monirul I Sharif, Andrea Lanzi, Jonathon T Giffin, and Wenke Lee. Impeding malware analysis using conditional code obfuscation. In *NDSS*, 2008.

[134] Michael Sikorski and Andrew Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, 2012.

[135] skylot. skylot/jadx: Dex to java decompiler, 2019.

[136] Tim Stazzere. Detecting pirated and malicious android apps with apkid, 2016.

[137] Arthur W. Stephens. Alas for adware - we know it too well. Technical report, The SANS Organization, 2005.

[138] Patrick Stewin and Iurii Bystrov. Understanding dma malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 21–41. Springer, 2012.

[139] 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*, pages 309–320. ACM, 2017.

[140] Guillermo Suarez-Tangil and Gianluca Stringhini. Eight years of rider measurement in the android malware ecosystem: evolution and lessons learned. *arXiv preprint arXiv:1801.08115*, 2018.

[141] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[142] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. The evolution of android malware and android analysis techniques. *ACM Computing Surveys (CSUR)*, 49(4):76, 2017.

[143] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. Data mining cluster analysis: basic concepts and algorithms. *Introduction to data mining*, pages 487–533, 2013.

[144] SophosLabs Research Team et al. Emotet exposed: looking inside highly destructive malware. *Network Security*, 2019(6):6–11, 2019.

[145] Ke Tian, Danfeng Yao, Barbara G Ryder, and Gang Tan. Analysis of code heterogeneity for high-precision classification of repackaged malware. In *Security and Privacy Workshops (SPW), 2016 IEEE*, pages 262–271. IEEE, 2016.

[146] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.

[147] Michel van Eeten, Johannes M Bauer, John Groenewegen, and Wolter Lemstra. The economics of malware. 2007.

[148] VirusTotal. Virustotal, 2019.

[149] John Von Neumann and A Burks. Theory of self-replicating automata. *Urbana: University of Illinois Press*, 1966.

[150] Haoyu Wang, Zhe Liu, Jingyue Liang, Narseo Vallina-Rodriguez, Yao Guo, Li Li, Juan Tapiador, Jingcun Cao, and Guoai Xu. Beyond google play: A large-scale comparative study of chinese android app markets. In *Proceedings of the Internet Measurement Conference 2018*, pages 293–307. ACM, 2018.

[151] Ting Wang, Shicong Meng, Wei Gao, and Xin Hu. Rebuilding the tower of babel: towards cross-system malware information sharing. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pages 1239–1248. ACM, 2014.

[152] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

[153] 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*, pages 252–276. Springer, 2017.

[154] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*, pages 62–69. IEEE, 2012.

[155] Victor Wu, Raghavan Manmatha, and Edward M. Riseman. Textfinder: An automatic system to detect and recognize text in images. *IEEE Transactions on pattern analysis and machine intelligence*, 21(11):1224–1229, 1999.

[156] 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*, pages 288–302. ACM, 2017.

[157] Kim Zetter. *Countdown to Zero Day: Stuxnet and the launch of the world's first digital weapon*. Broadway books, 2014.

[158] 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 *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, CODASPY '15, pages 37–48. ACM, 2015.

[159] 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*, pages 185–196. ACM, 2013.

[160] 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*, pages 317–326. ACM, 2012.

[161] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012.

[162] Shuofei Zhu, Jianjun Shi, Limin Yang, Boqin Qin, Ziyi Zhang, Linhai Song, and Gang Wang. Measuring and modeling the label dynamics of online anti-malware engines. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

Glossary

AI Artificial Intelligence. 54

API Application Programming Interface. 5, 21, 24, 25, 30, 35, 36, 38, 53, 55–58, 60, 70, 83, 86, 87, 95, 104, 106, 108, 130, 138–140, 142, 143, 160, 165, 191, 195

APK Android Package. 4, 5, 11, 26, 41, 43, 53–58, 60–62, 65, 66, 69, 70, 106, 130, 133, 139, 140, 142, 151, 199

ART Android Runtime. 26

AVD Android Virtual Device. 97, 139, 140, 151, 195

CFG Control Flow Graph. 3, 23

CRUD Create, Read, Update, and Delete. 24

CSV Comma-Separated Values. 151

DCL Dynamic Code Loading. 31, 32, 148, 149

DLL Dynamic-link Library. 26

DMA Direct Memory Access. 147, 148

EM Expectation Maximization. 156

GNB Gaussian Naive Bayes. 65, 130, 133, 134, 189

GPS Global Positioning System. 29, 36, 56, 120, 167

GUI Graphical User Interface. 22, 24, 25, 83, 141–143, 148

HMM Hidden Markov Model. 5, 54

IDE Integrated Development Environment. 25, 32, 54

IMEI International Mobile Equipment Identity. 29, 38, 148

IO input-output. 52

IP Internet Protocol. 36, 57

JAR Java ARchive. 26

JDK Java Development Kit. 25

JSON JavaScript Object Notation. 58, 140

KNN K-Nearest Neighbors. 62, 130, 133, 134, 189

MAC Media Access Control. 38

MCC Matthews Correlation Coefficient. 77–82, 84, 86, 87, 92–94, 106, 113, 115, 117, 122, 123, 126, 128–134, 167, 188, 189, 193

MD5 Message Digest. 106

ML Machine Learning. ix, xi–xv, 3–16, 18–20, 51–55, 61–66, 68–71, 89, 90, 92, 95, 98, 104–109, 111–117, 119–135, 151, 152, 154, 156, 157, 159, 160, 163, 164, 167–170, 187–189, 193, 198, 201

NDK Native Development Kit. 25

OS Operating System. 38, 140, 148

PCA Principal Component Analysis. 61

PUA Potentially Unwanted Application. 30

RF Random Forest. 65, 130, 133, 134, 189

SDK Software Development Kit. 24–26, 54

SHA Secure Hash Algorithms. 106

SIM Subscriber Identification Module. 36

SMS Short Message Service. 23, 29, 37, 40, 47, 69, 148, 149

SQL Structured Query Language. 23

SSIM Structural Similarity Index. 143

SVC Linear Support Vector Machine. 131

SVM Support Vector Machine. 5, 6, 54, 65, 67, 130, 187

TTP Tactics, Techniques, and Procedures. 53

URL Uniform Resource Locator. 32, 56, 57, 153

VM Virtual Machine. 26

XML Extensible Markup Language. 24–26

List of Figures

Listings

List of Tables

Appendix

A. Manual Analysis Process

The following steps depict the process we adopted to manually analyze and label apps in the *Hand-Labeled* and *Hand-Labeled 2019* datasets. Given an app (α), we:

1. Install (α) on a rooted AVD containing the `Xposed` framework and the API call monitoring tool `droidmon`.
2. Run and interact with (α) on the AVD and monitor the API calls it issues during run time.
3. If app crashes or no malicious behavior is noticed: decompile (α) with `Jadx` and inspect its source code.
4. If no traces of malicious code are found, disassemble (α) with `Apktool` and reverse engineer its `smali` code.
5. If no traces of malicious code are found, disassemble the shared object libraries used by (α) using `Gidhra` and inspect `C/C++` code.
6. If no traces of malicious code are found, provisionally deem an app as benign and check `VirusTotal` regarding the app's status.
7. If (α)'s `VirusTotal` scan report gives a total of *zero* positives, deem app as benign. Otherwise, inspect the scanners deeming app as malicious, the labels they give to the app (e.g., `Riskware`), and the details inside the scan report.
8. If the aforementioned details reveal malicious behavior, deem as such. Otherwise, deem app as benign.

B. `BitDefender` and `Panda` Vs. AMD Apps

The `SHA1` hashes of ten apps randomly sampled from the *AMD* dataset, the number of scanners deeming them as malicious as of September 27^{th}, 2019, their malware types, and whether `BitDefender`'s version 3.3.063 and `Panda`'s version 3.4.5 managed to detect them.

SHA1 Hash	VirusTotal *positives*	Malware Type	BitDefender	Panda
fdf0835597adc16d667b4cbaef0fc3ee76205503	22	Adware		
3425e68789614cbda4284169589f7aeab8e28b28	29	Trojan-SMS	✓	✓
11c8678985aaa5ce4bcd5970a0008b0310c88a7e	35	Trojan-SMS	✓	✓
7693726e8f286d6dd8c115c273637d44c554f19c	28	Trojan-Banker	✓	
a0e7e36eec83339980f056a7af5f36d5dc99809e	33	Ransom	✓	✓
74581e2e0c7b93391574ec8582d274432a4a2838	27	Adware		
a9a46d346a2ae5a397517c70d249695c6c89632b	19	Adware		
db9603ffa852f1b0aad3f44418db44893db3f726	29	Adware	✓	
7f9e6bc600b4a02b5d15a6511a43419d1aa500ff	29	Adware	✓	
d4eb21ae5c2b4b05c0f3ce4e5117c56d9c3746ef	23	Adware	✓	

C. Maat's Engineered Features

The following list enumerates the engineered features extracted from the VirusTotal scan reports of apps in the *AMD+GPlay* dataset. The order of features in the list mimics the order of every feature in the feature vector.

1. The vedicts (i.e., 1 for malicious, 0 for benign, and -1 for unknown), given by the intersection of the most correct and stable VirusTotal based on the scan reports of apps in the *AMD+GPlay* dataset between November 2018 and September 27th, 2019:
   - AhnLab-V3
   - Avira
   - Babable
   - CAT-QuickHeal
   - Comodo
   - Cyren
   - DrWeb
   - ESET-NOD32
   - Fortinet
   - Ikarus
   - K7GW
   - MAX
   - McAfee
   - NANO-Antivirus
   - Sophos
   - SymantecMobileInsight
2. The age of a scan report in years calculated as the difference between today's date and that of *first_seen*'s.

3. The number of times the app was submitted to `VirusTotal` according to the *times_submitted* attribute.
4. The number of scanners deeming the app as malicious according to the *positives* attribute.
5. The total number of scanners that scanned the app according to the *total* attribute.
6. The list of permissions requested by the app out of 324 permissions. If a permissions is requested by an app, its corresponding index in the feature vector has a value of 1, and a value of 0 otherwise.
7. The list of tags given by `VirusTotal` to the app out of 32 tags. If a tag is given to an app, its corresponding index in the feature vector has a value of 1, and a value of 0 otherwise.

D. Maat's Selected Naive Features

The following list enumerates the *selected* naive features, viz. the verdicts of `VirusTotal` scanners that train the best performing ML-based labeling strategies. The order of features in the list mimics the order of every feature in the feature vector.

- `AhnLab-V3`
- `Avira`
- `CAT-QuickHeal`
- `Cyren`
- `DrWeb`
- `ESET-NOD32`
- `F-Secure`
- `Fortinet`
- `Ikarus`
- `K7GW`
- `MAX`
- `McAfee`
- `McAfee-GW-Edition`

- NANO-Antivirus
- Sophos
- SymantecMobileInsight
- Trustlook

E. Maat's Hyperparameter Estimation

To train the best random forests that constitute the ML-based labeling strategies, Maat uses the techniques of *grid search* and *random search* to estimate the hyperparameters of the trees in those forests. We used 10-Fold Cross Validation to train random forests of 100 decision trees and varied the following parameters as follows:

- The criterion used to choose the feature to check to further split the training dataset into malicious and benign apps `criterion`: $\{gini, entropy\}$
- The maximum depth a decision tree is allowed to grow `max_depth`: $\{1, 4, 10, None\}$
- The maximum number of features a decision tree is allowed to check upon every split `max_features`: $\{3, 5, 10, None\}$
- The minimum number of samples required to split a node in a tree `min_samples_split`: $\{2, 3, 10\}$
- If False, the entire dataset is used to train the decision tree instead of bootstrap samples `bootstrap`: $\{True, False\}$

F. Homegrown Dataset

The `SHA1` hashes, a short description, and the `VirusTotal` *positives*:*total* fields of apps in the *Homegrown* dataset, as of November 8^{th}, 2019.

`SHA1` Hash	Description	`VirusTotal` *positives*	`VirusTotal` *total*
17866baec8c1179264c585934d742a7befa20975	Encryption-based Ransomware demo app	0	56
1b8235f2ad665df5f8632b75a1b466f849654934	Tapjacking-based tracking app and WiFi password stealer	0	56
1c7c935ba48c5db86ff4fd957fedc3e691484c77	Tapjacking and overlay-based Ransomware demo app	0	56
31cf1a7a7f8a3bb6f9fdb45267dcbf9ff449b994	Repackaged app with encryption-based Ransomware payload	0	56
66c16d79db25dc9d602617dae0485fa5ae6e54b2	Repackaged app with logic-based payload to delete user contacts	1	56
691973efb45176862085e4d4e081dfa8750590f7	Repackaged antiviral software with backdoor	0	55
aa0d0f82c0a84b8dfc4ecda89a83f171cf675a9a	Repackaged mail client with obfuscated encryption-based Ransomware	0	52
bee87f0ae97b438488cbe351311d5d40ccf8c3e0	Launcher-based Ransomware demo app	0	56

G. Static Features

The following list enumerates the numerical features statically extracted from the APK archives of apps in the *AndroZoo*, *Hand-Labeled*, and *Hand-Labeled 2019* datasets. The order of features in the list mimics the order of every feature in the feature vector.

- **Basic features**:
  - (1) Minimum SDK version (supported by the app).
  - (2) Maximum SDK version (supported by the app).
  - (3) Total number of activities.
  - (4) Total number of services.
  - (5) Total number of broadcast receivers.
  - (6) Total number of content providers.
- **Permission-based features**:
  - (7) Total request permissions.
  - (8) Ratio of Android permissions to total permissions.
  - (9) Ratio of Custom permissions to total permissions.
  - (10) Ratio of dangerous permissions to total permissions.
- **API call features**:
  - (11) Total number of classes.
  - (12) Total number of methods.
  - (13)-(39) Count of calls to methods in the following packages:
    - `android.accounts.AccountManager`
    - `android.app.Activity`
    - `android.app.DownloadManager`
    - `android.app.IntentService`
    - `android.content.ContentResolver`
    - `android.content.ContextWrapper`
    - `android.content.pm.PackageInstaller`
    - `android.database.sqlite.SQLiteDatabase`
    - `android.hardware.Camera`
    - `android.hardware.display.DisplayManager`

  - `android.location.Location`
  - `android.media.AudioRecord`
  - `android.media.MediaRecorder`
  - `android.net.Network`
  - `android.net.NetworkInfo`
  - `android.net.wifi.WifiInfo`
  - `android.net.wifi.WifiManager`
  - `android.os.PowerManager`
  - `android.os.Process`
  - `android.telephony.SmsManager`
  - `android.widget.Toast`
  - `dalvik.system.DexClassLoader`
  - `dalvik.system.PathClassLoader`
  - `java.lang.class`
  - `java.lang.reflect.Method`
  - `java.net.HttpCookie`
  - `java.net.URL.openConnection`

- **Miscellaneous features**:

  (40) Zero-based index of the compiler used to compile the app from (`dx`, `dexmerge`, `dexlib 1.x`, `dexlib 2.x`, `Jack 4.x`, or unknown).

H. Papers Usage and Author Contribution

In this section, we enumerate the peer-reviewed papers that has been used in this doctoral thesis, how and where in the thesis they have been used, and the role of the author of this thesis in writing those papers.

- **Salem, A.**; Paulus, F.; Pretschner, A. *Repackman: A Tool for Automatic Repackaging of Android Apps*. In Proceedings of the 1st International Workshop on Advances in Mobile App Analysis (A-Mobile), 2018.
  - Purpose and Location of Use: Content from this tool paper has been **copied** and **used** in Section 2.4.6 to demonstrate one method of repackaging Android apps with malicious payloads.

  - Thesis Author's Role: The author of this thesis wrote the tool paper with the assistance of the paper's second author, building on her bachelor's thesis, which she completed at the chair of software and systems engineering led by the third author. Based on the second author's thesis, the first author wrote the paper, and the second author helped with reviewing the paper's structure, content, and with creating a video that demonstrates the tool's functionalities.
  - Other Author(s): At the time of writing this paper, F. Paulus was a bachelor of informatics student and A. Pretschner was the first author's doctoral supervisor.

- **Salem, A.**; Pretschner, A. *Poking the Bear: Lessons Learned from Probing Three Android Malware Datasets*. In Proceedings of the 1st International Workshop on Advances in Mobile App Analysis (A-Mobile), 2018.
  - Purpose and Location of Use: The content in Section 2.2 and Section 2.3, which describe the structure of Android malware and the types of malicious payloads they use, is **based** on the findings of the paper in question.
  - Thesis Author's Role: The author of this thesis conducted the experiments found in the paper and wrote the paper under the guidance and with the help of the second author.
  - Other Author(s): At the time of writing this paper, A. Pretschner was the first author's doctoral supervisor.

- **Salem, A.**; Schmidt, T.; Pretschner, A. *Idea: Automatic Localization of Malicious Behaviors in Android Malware with Hidden Markov Models*. In Proceedings of the International Symposium on Engineering Secure Software and Systems (ESSoS), 2018.
  - Purpose and Location of Use: The content and idea of this paper serves, in Section 3.3, as an example of Android malware detection methods that combine dynamic representations of Android apps with ML-based detection. No material from this paper has been copied or used in this doctoral thesis.
  - Thesis Author's Role: The author of this thesis wrote the paper with the assistance of the paper's second author, building on her master's thesis, which she completed at the chair of software and systems engineering led by the third author.
  - Other Author(s): At the time of writing this paper, T. Schmidt was a bachelor of informatics student and A. Pretschner was the first author's doctoral supervisor.

- **Salem, A.**; Hesse, M., Neumeier, J., Pretschner, A. *Towards Empirically Assessing Behavior Stimulation Approaches for Android Malware* . In Proceedings of the 13th International Conference on Emerging Security Information, Systems and Technologies (SECURWARE), 2019.
  - Purpose and Location of Use: The content in Section 2.4 **builds** on the types of triggers used by malware authors to delay/control the execution of malicious payloads in Android malware, whereas the content in

  - Thesis Author's Role: The author of this thesis used the tools set up by the second author during his employment at the chair of software and systems engineering and built on the third author's bachelor's thesis to conduct the experiments found in the paper and write it under the guidance of the fourth author.
  - Other Author(s): At the time of writing this paper, M. Hesse was a research assistant studying towards his bachelor's of informatics and J. Neumeier was a bachelor of informatics student, and A. Pretschner was the first author's doctoral supervisor.

- **Salem, A.**; Banescu, S., Pretschner A. *Maat: Automatically Analyzing VirusTotal for Accurate Labeling and Effective Malware Detection*. In ACM Transactions on Privacy and Security (TOPS) **[Under Review]**, 2021.
  - Purpose and Location of Use: This paper summarizes the main contribution of this doctoral thesis (i.e., the platform Maat), and discusses the insights gained from the measurements and experiments we conducted. Content of this paper is **copied** from different chapters in this thesis, namely Chapter 4, Chapter 5 and Chapter 6.
  - Thesis Author's Role: The author of this thesis conducted the aforementioned measurements and experiments under the guidance of the second and third authors, who helped in writing the paper as well.
  - Other Author(s): At the time of writing this paper, S. Banescu was a post-doctoral researcher at the chair of software and systems engineering and A. Pretschner was the first author's doctoral supervisor.

- **Salem, A.** *Towards Accurate Labeling of Android Apps for Reliable Malware Detection*. In Proceedings of the 11th ACM Conference on Data and Application Security and Privacy (CODASPY), 2021.
  - Purpose and Location of Use: This paper focuses on the impact of `VirusTotal`'s dynamicity on the threshold-based labeling strategies that are commonly used within the research community. The content of this paper is **based** on the content in Chapter 4 and Chapter 7.
  - Thesis Author's Role: The author of this thesis conducted the experiments found in the paper and wrote the paper under the guidance and with the help of the second author.
  - Other Author(s): At the time of writing this paper, A. Pretschner was the first author's doctoral supervisor.