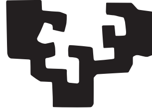


Konputagailuen Arkitektura eta Teknologia Saila

Departamento de Arquitectura y Tecnología de Computadores

Artificial Intelligence-based contributions to the detection of threats against information systems

by

Borja Molina Coronado

Supervised by Usue Mori and Jose Miguel-Alonso

Donostia - San Sebastián, October 2023

(cc)2023 BORJA MOLINA CORONADO (cc by-nc 4.0)

Acknowledgments

First of all, I would like to thank my advisors Usue Mori and Jose Miguel-Alonso, as well as to Alexander Mendiburu that has been in the shadows, for their help. Without their guidance, this dissertation would never have been possible. I also owe a debt of gratitude to Prof. Alessio Merlo for allowing me to complete a stay as a visitor in his research group at the University of Genoa. I also would like to acknowledge the great work of John Kennedy, whose comments on English writing have improved the readability of the published work product of this thesis.

I am grateful to all the people that, to a lesser or a greater extent, has relied on me throughout this long period. I am lucky to have shared most of my time during this thesis with great people. When I moved to Cuenca I knew I left part of my family behind. Special thanks go to the Pio Baroja's climbing crew, my flatmate Ander and Goiatz, who reversed my experience. I cannot fail to mention the people from the Faculty of Computer Science. *Txalupas*, *Trader*, *Vinos*, *Imalcohol*, *Vanidoso* and *Peli*, all of you did this period way easier, and the moments we lived together will always remain in my mind and make me smile when remembered.

The main pillars of my life mere to be mentioned. Though it was not easy, they guided me from distance, making me feel them very close. Thank you Papá, Mamá and Rubén. Beyond me, this thesis is for Irene, whose strength keeps fascinating me, and our future daughter, Vega.

Contents

1 **Introduction** 1

Part I On the use of AI algorithms for network intrusion detection

2 **A KDD approach to Network Intrusion Detection** 7
- 2.1 Introduction 7
- 2.2 Basic Concepts 10
- 2.3 Data Collection Stage 13
- 2.4 Preprocessing: Derivation of Features 16
- 2.5 Preprocessing: Data Cleaning and Feature Transformation 24
- 2.6 Data Reduction 28
- 2.7 Data Mining Stage 32
- 2.8 Evaluation Stage 45
- 2.9 Discussion and Conclusions 48

Part II On the use of AI algorithms for Android malware detection based on static analysis

3 **Background** 55
- 3.1 Android Apps 55
- 3.2 Supervised Classification for Android Malware Detection 58
- 3.3 App Obfuscation in Android 59

4 **Toward realistic AI-based Android malware detection** 63
- 4.1 Introduction 63
- 4.2 Related Work 66

4.3 Android Malware Detectors . . . 67
4.4 Datasets. Drawbacks and Reproducibility Issues . . . 69
4.5 Experimental Setup . . . 73
4.6 Comparative Analysis . . . 75
4.7 Discussion: Towards a Realistic Framework for Malware Detection . 86
4.8 Conclusions . . . 87

5 Dealing with obfuscation . . . 89
5.1 Introduction . . . 89
5.2 Related Work . . . 91
5.3 Dataset . . . 93
5.4 Feature Validity . . . 97
5.5 Robust Malware Detection . . . 106
5.6 Discussion . . . 108
5.7 Conclusions . . . 110

6 Dealing with concept drift . . . 111
6.1 Introduction . . . 111
6.2 Related Work . . . 113
6.3 Preliminary Concepts . . . 115
6.4 Retraining Frequency . . . 117
6.5 Data Used for Retraining . . . 118
6.6 Experimental Framework . . . 122
6.7 Experimental Results . . . 125
6.8 Analysis of the Combined Effect of Change Detection and Sample Selection Methods . . . 130
6.9 Conclusions . . . 130

Part III Summary of Main Findings

7 Conclusions and Future work . . . 135
7.1 Contributions . . . 135
7.2 Future work . . . 137
7.3 Publications . . . 140
7.4 Data and Code . . . 140

References . . . 141

No importa la tierra que pisas,
sino la huella que dejas

1

Introduction

In contemporary society, the widespread integration of technology into our daily routines has led to a significant reliance on digital infrastructure. This shift is marked by the prevalence of information systems, which play a pivotal role for human interaction, data exchange, and technological progress [97]. This era of extensive connectivity, known as the Big Data era, is characterized by the substantial creation, transmission, and storage of data. Such increasing volume of data requires not only innovative approaches to data analysis and management but also practical solutions to safeguard the integrity, confidentiality, and availability of the underlying information systems.

In a context where manual analysis is not feasible due to the variety, velocity, and volume of data, Artificial Intelligence (AI) has emerged as an exceptional technological advancement. AI, and more specifically machine learning (ML) algorithms, possesses the capability to rigorously analyze and safeguard information systems from the risks they face. At its core, AI, offers a feature of paramount importance in the realm of cybersecurity: the ability to uncover intricate patterns within data that would otherwise be prohibitively costly and time-consuming for human experts to extract. This characteristic makes AI a pivotal tool in Alert Management, Vulnerability and Risk Management, and Threat Detection [18].

In alert management systems, AI systems can assist human analysts to alleviate them from the task of sifting through an overwhelming flow of alerts, guaranteeing that only genuine security incidents are escalated for further investigation. The remarkable ability of AI not only has helped to differentiate false alarms from authentic security threats [33] but also, has elucidated a future for automatic incident response systems [156].

Vulnerability and risk management is another critical area of cybersecurity that traditionally relied heavily on manual efforts. In this area, the adoption of AI techniques for identifying software vulnerabilities has significantly improved efficiency, coverage and accuracy, expanding the identification of potential software vulnerabilities that could expose organizations and users to security threats [146]. Furthermore, AI-based proposals that assist in implementing remediation measures are also emerging [300].

For threat detection, AI has the potential to identify signs of attacks in real-time, enabling security operators to take measures that minimize their impact [125]. Specifically, AI has served as a proactive guardian against spam, which is a common medium for distributing phishing campaigns and propagating malware [119]. In the realm of intrusion detection [207] and malware detection, with its incorporation, AI has contributed to significant transformations, changing the detection paradigm from the definition of static rules to the automatic extraction of behavioral patterns of attacks and malware, at speeds that surpass the capabilities of human experts [289].

It should be noted, however, that the application of AI to solve cybersecurity problems is not free of challenges. On the one hand, cybersecurity solutions operate in a highly complex and dynamic environment where threats and attack techniques against information systems are constantly evolving. The ever-changing nature of this landscape requires continuous adaptation and updates of AI strategies and models [139, 208]. Moreover, AI solutions heavily rely on data for training and decision-making and the lack of access to comprehensive and diverse datasets can hinder the effectiveness of AI solutions. The obtention of high-quality, labeled data can be very difficult in some cybersecurity problems, where data is often noisy, incomplete, and subject to privacy and regulatory constraints [271].

On the other hand, by its hostile nature, attackers are continuously implementing more sophisticated adversarial strategies to evade detection. They can use AI to attack and manipulate security solutions, making them produce incorrect results or overlook vulnerabilities [325]. Since AI is a rapidly evolving field, this arms race between AI-based defenses and adversarial attackers requires constant research and development.

Since the proposal of effective AI-based cybersecurity solutions must aim to solve the aforementioned challenges, the evaluation of these methods should take these aspects into account in order to provide credible and useful results. However, this is not common practice and has given rise to controversies regarding the utility of AI in the field of cybersecurity. In this context, some authors have highlighted the lack of rigour and attempted to enumerate the most common deficiencies in the proposal and evaluation of AI techniques in the scientific literature on cybersecurity [23]. Among the most prevalent deficiencies are the simplification of experimental evaluation conditions and the omission of data, code, or details necessary for the reproduction of AI-based proposals, leading to unrealistic results and generating distrust in AI. Nevertheless, these practices should not overshadow the potential of AI, which has indeed taken significant steps toward enhancing the cybersecurity landscape.

The aim of this thesis is to analyze the potential of AI methods to address information security problems in the area of threat detection. Specifically, we focus our study into Network Intrusion Detection Systems (NIDS), which represent the first line of defence against attacks and address the identification of treats targeting information systems through the examination of network activity; and on malware detection for Android, which is based on the analysis of the characteristics of apps to identify malware.

Along the first part of the dissertation (Part I), we review and analyze research proposals on the area of NIDS presented until 2019 from a perspective of the Knowledge Discovery in Databases (KDD) process. KDD serves as a guideline to discuss the techniques applied to network data from its capture to the detection of attacks using AI methods, focusing on their applicability to real-world scenarios. As a result of this thorough review process, we identify several open issues which need to be considered for further research in the area of network security.

The second part of this dissertation (Part II) focuses on contributions to malware detection in the Android operating system, specifically on detectors that leverage static analysis information from apps. In contrast to dynamic analysis, static analysis does not require executing the app. Chapter 3 introduces fundamental concepts of Android malware detection using static analysis, providing essential context for the rest of this dissertation. Chapter 4 identifies five realistic experimental factors present in production scenarios that are often overlooked in the scientific literature. In addition, we propose a fair evaluation framework for malware detectors aimed at providing a standardized experimental methodology, something that currently does not exist. In particular, our findings reveal that most detectors struggle when evaluated under certain conditions. Subsequent chapters aim to provide more realistic alternatives or improvements to these detectors by addressing some of the pitfalls evidenced. In Chapter 5, we present a study examining the validity of static analysis features against obfuscation, a technique frequently employed by malware developers to evade detection. Finally, in Chapter 6, we explore efficient methods to adapt to the continuously evolving nature of the malware detection problem, particularly for AI-based malware detectors that do not address this aspect by design.

The last part of this dissertation (Part III) draws the general contributions and conclusions that can be extracted from this work, and points out some potential future lines of research in the NIDS and Android malware detection areas.

Part I

On the use of AI algorithms for network intrusion detection

2

A KDD approach to Network Intrusion Detection

Network intrusion detection is a complex problem that presents a diverse range of challenges. The first part of this dissertation focuses on studying techniques aimed at identifying network attacks targeting information and communication systems using AI techniques such as machine learning (ML) algorithms. We provide an analysis of these methods from the perspective of the Knowledge Discovery in Databases (KDD) process. Accordingly, we will discuss the techniques used for data collection, preprocessing, and data transformation, as well as the ML algorithms and evaluation methods employed. We will also elaborate on the characteristics and motivations behind the use of each of these techniques and propose more accurate and up-to-date taxonomies and definitions for intrusion detectors, drawing from terminology commonly used in the field of data mining and KDD. Special attention will be given to the evaluation procedures used to assess the detectors, along with their applicability in current, real-world networks. All of these aspects will shed light on open issues that require consideration for further research in the field of network security.

2.1 Introduction

Intrusion Detection Systems (IDSs) are deployed to uncover cyberattacks that may harm information systems. NIDS operate with network-related data, focusing exclusively on network traffic analysis, with the aim of finding malicious patterns targeting all the machines or devices inside the protected network [130].

Finding attack indicators in network traffic data presents important particularities [271, 141] such as (1) the growing number of attacks, which arise and change as new vulnerabilities are identified; (2) the diverse and evolving nature of network traffic; (3) the continuously growing data volume, which challenges the successful analysis of network traffic; and (4) the increasingly common use of encrypted data, not easily accessible to detectors. These facts, along with the need for near real-time responses, make intrusion detection a very hard problem.

Data mining techniques, which are applied successfully in many fields, have also proven useful to implement NIDS. These techniques can find complex relationships in the data, which can be used to detect network attacks. However, off-the-shelf data mining algorithms cannot be applied directly to network data, as they cannot cope with the particularities listed above. Intrusion detection is the result of a complex process that starts with the collection of network data and continues with the preparation and preprocessing of this data. Only then will a detector be able to deliver meaningful results.

In spite of this, NIDS research normally focuses only on the application of data mining algorithms to network datasets, paying minimum attention (or no attention at all) to questions such as: Which type of data (raw traffic, flows, encrypted/-plaintext/no payload, etc.) will the NIDS receive? Can all the data traversing the network be gathered and processed, or just an excerpt of it? Where will the NIDS be located? How good is the performance of the NIDS in terms of both accuracy and response time? Can the detection model be updated easily to detect new threats? In this chapter we aim to provide some answers to these questions.

Survey	Data collection	Data Preprocessing and Transformation	Data mining	Detector updates
Garcia et al., (2009) [105]			✓*	
Sperotto et al., (2010) [274]	✓*		✓*	
Davis et al., (2011) [71]	✓	✓*		
Bhuyan et al., (2014) [41]			✓*	
Ahmed et al., (2015) [6]			✓*	
Buzcak et al., (2016) [53]			✓	
Umer et al., (2017) [290]			✓*	
Moustafa et al., (2019) [211]	✓*	✓*	✓*	
Khraisat et al., (2019) [160]	✓*		✓	
This Survey	✓	✓	✓	✓

*Partial coverage

Table 2.1: Summary of NIDS surveys and their coverage of the different components that conform a functional NIDS.

Several surveys of NIDS research have been published already. Table 2.1 summarizes their scope. Most of them include only a partial coverage of the steps that are necessary to perform intrusion detection by means of network traffic analysis. As can be seen in the table, most published NIDS surveys overlook data collection [104, 41, 6, 53, 290, 160], while others cover it but only partially, either providing descriptions that are too shallow [211], or focusing on a subset of the possible sources of data [274]. Data preprocessing and transformation phases have received a similar lack of analysis: the number of surveys covering these tasks is small, with limited content and they ignore their associated problems and challenges [71, 211].

In contrast, the data mining phase has been the focus of almost all NIDS surveys. An extensive description of a variety of methods that constitute the “core” of NIDS (the detection engine) is given in [53, 160]. Some papers also focus on this phase but for particular detector types. For example, surveys [105, 41, 6, 211] review only detection techniques based on the discovery of anomalous network traffic, while in [274, 290] the focus is only on approaches that use a specific type of input data,

ignoring detectors that use other data sources (such as raw packet data). Finally, we are not aware of any survey addressing detector updates that aim to deal with the evolving nature of traffic and attacks [292], an essential feature of a production NIDS.

We reckon that previous reviews transmit the idea that the only relevant part of a NIDS is the data mining algorithm selected or devised to identify attacks. However, NIDS actually comprise many processes related to KDD that have not been covered in enough detail, e.g., how data is extracted, preprocessed and transformed; or how the detection engine is evaluated and updated. These additional processes play a fundamental role in the detection ability of a NIDS, and deserve deeper analysis.

In this chapter, we review NIDS proposals using the Knowledge Discovery in Databases (KDD) process as a guideline. This chapter presents the following contributions:

- We describe the basic blocks that comprise a functional NIDS from the perspective of the KDD process, going beyond the selected data mining algorithm used for network attack detection.
- We review the techniques used in all the steps of this process, paying special attention to the motivation behind their use.
- We propose a taxonomy of NIDS detection methods based on the data mining terminology to avoid the ambiguity of previous classification proposals. To that end, we use two different criteria: (a) the detection approach and (b) the learning approach.
- We review the mechanisms to avoid deterioration in detection ability due to the evolution of network traffic, including attacks.
- We discuss NIDS validation and evaluation procedures, identifying common mistakes such as the use of inadequate metrics based on unrepresentative datasets.
- We discuss current challenges in NIDS research based on the surveyed literature. We analyze the processes and steps followed by authors to build NIDS, identifying common assumptions and mistakes, and highlighting future research lines in the area of network security.

This chapter is organized as follows. First, we briefly depict some basic concepts about how network attacks can manifest themselves and describe the KDD process and its phases. In the following sections we enumerate and categorize the most representative techniques proposed in the NIDS literature for the different phases of the KDD process. We start with the collection of network traffic data. In Sections 2.4 and 2.5, we analyze methods for the extraction of structured feature records from that data, as well as the transformations that are commonly applied to those features. Techniques to remove data redundancies and accelerate the application of data mining algorithms are discussed in Section 2.6. In Section 2.7 we pay attention to the data mining phase, analyzing different learning methods and detection approaches. In Section 2.8 we show the evaluation criteria used by researchers, and also discuss their suitability for NIDS. This panoramic view of NIDS proposals allows us to close the chapter in Section 2.9 discussing the challenges

in NIDS research, identifying common mistakes and unrealistic assumptions in the network security area.

2.2 Basic Concepts

In this section we introduce network attacks (those that a NIDS must detect) and the way they manifest if network traffic is analyzed.

2.2.1 Networks Under Attack

The CIA triad (Confidentiality, Integrity and Availability) is a widely known set of security properties used in information security management for the definition of security policies and procedures [202]. Confidentiality refers to the efforts to keep data private or secret, controlling access to prevent unauthorized disclosure. Integrity is about guaranteeing that data has not been altered and, therefore, can be trusted: it is correct, authentic and reliable. Availability measures seek to ensure that authorized users have timely and reliable access to resources when they are needed. From the perspective of information security, any attempt to compromise any of the policies and mechanisms set up by an organization to guarantee the CIA properties of a system can be considered an attack [95].

When an attacker uses the organization's network to reach their target, the corresponding action is considered a *network attack* [126]. In this context, the attacker is also known as an *intruder*, and the action as an *intrusion.* An analysis of the traffic managed by the network should help to detect network attacks by finding the leads that the intrusions have left behind. This is precisely what NIDS do: they are fed with network data and search for clues that may uncover malicious activities. There is a large variety of attacks, with different targets and purposes, and the clues and symptoms they leave are not always obvious. Table 2.2 lists some broad groups of known network attacks.

As stated before, network intrusion detection is only possible if network attacks leave evident or latent traces in the data collected by the NIDS. These traces may be visible only from some specific vantage points and may appear at different traffic inspection levels [287][1]. Attack manifestations can be classified as:

- **Point**. The malicious activity affects one data sample, e.g., a packet, a connection. In this case, an attack leaves a footprint in a single record, meaning that some of their values are not permitted or constitute an anomaly. For example, SQL injection is carried out by placing code in the parameters of a HTTP request with the aim of manipulating the behavior of a web application. This attack can be detected by analyzing a single sample of HTTP payload.

[1] We use the classification made by Thottan et al. [287] to introduce how attacks become apparent at different traffic inspection levels. However, different alternative taxonomies are available, see [202], [126] and [138].

Attack	Confidentiality	Integrity	Availability	Description
Brute force	✓			Repetitive attempt to carry out some action, such as authentication or resource discovery, often guided by a dictionary.
Injection (SQL, command...)	✓	✓		Exploitation of weaknesses in input field sanitizing mechanisms to execute code or commands on the target system.
Privilege escalation	✓			Attempt to gain unauthorized access from an unprivileged account.
Spoofing	✓	✓		A person or program masquerades as another, falsifying data to gain an illegitimate advantage.
Sniffing	✓			Eavesdropping traffic in order to obtain information.
(Distributed) denial of service			✓	Resource abuse intended to deny access to a service to its legitimate users.

Table 2.2: Examples of broad classes of network attacks, indicating the information security properties (CIA) they compromise.

- **Contextual**. A particular event is not an anomaly in itself, but may indicate that an attack is happening due to its unexpected appearance given the context. For example, a HTTP (web) request to a DNS server is not an anomaly by itself, but could be indicative of a service discovery attempt over a server.
- **Collective**. The attack shows up as several characteristic events which, together but not individually, constitute a malicious event. For example, a large number of simultaneous requests to a common service form an anomaly, although each individual connection is not anomalous by itself.

Therefore, given the diverse forms in which a malicious event may leave a footprint, different strategies need to be taken during the construction of a NIDS. The successful identification of network attacks is not only related to the choice of a good data mining algorithm, but also to the information available (collected and prepared) to feed it.

2.2.2 The Knowledge Discovery in Databases Process

The Knowledge Discovery in Databases (KDD) process describes the procedures commonly used to find explainable patterns in data, allowing the interpretation or prediction of future events [89]. KDD was defined formally in [234] as:

> *"KDD is the organized process of identifying valid, novel, useful, and understandable patterns from large and complex data sets"*

KDD is an iterative process, as depicted in Figure 2.1, comprising the following steps [89]:

- **Data collection** consists of capturing and selecting raw data from their sources. This data should provide the necessary information to solve the problem at hand.
- **Data preprocessing** involves the extraction of a valid and structured set of features required for knowledge extraction. It also includes the application of

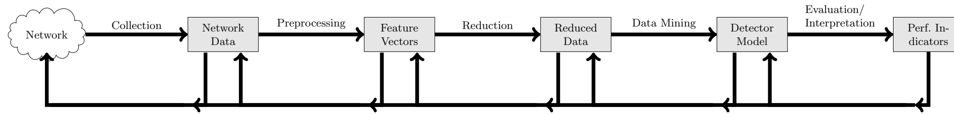


Fig. 2.1: Illustration of the KDD process for NIDS, showing its iterative nature, the results of each stage, and the possibility of going back and rethinking any previous stage.

cleaning methods to avoid factors such as noise, outliers or missing values, and the derivation of new features from others to structure the data in a way appropriate for the application of the algorithms of the following phases.

- **Data reduction** is a necessary step to optimize the application of data mining algorithms, reducing their computation time and/or improving their results. It involves feature subset selection (removal of useless or redundant features), dimensionality reduction (projection of a set of features into a smaller space) or modifications of the set of samples (adding or removing samples).
- **Data mining** is the core of the KDD process, but it can be successful only if the previous steps are performed properly. It consists of selecting and applying techniques, including Machine Learning (ML) algorithms, to extract knowledge from data by finding useful patterns and relations between features. The choice of a specific data mining algorithm depends on the problem at hand and the nature of the available data.
- **Interpretation and evaluation** consists of, using a set of metrics, measuring the performance and effectiveness of the data mining algorithm, and, consequently, of the tasks carried out in previous steps of the KDD process. It also comprises result visualization and reporting.

Given the iterative nature of this process, some steps may have to be reapplied and rethought more than once to achieve satisfactory results [79]. For example, a feature initially excluded during the data transformation step may be added later in order to increase the performance of a particular data mining algorithm.

In the NIDS context, the KDD process starts with the collection of network data. Preprocessing is essential to organize the large and diverse set of collected data, transforming it into a set of structured records with a common set of features. Data mining can be applied to this collection of records, or to a reduced version (after some data reduction procedures), and then evaluated using adequate metrics: detection rate, false alarm rate, etc. With some variations, this should be the NIDS pipeline. However, most reviewed papers start with publicly available datasets, already collected and preprocessed. An advantage of this approach is the reduction in the number of tasks (focusing mostly on the data mining part), and the availability of a common evaluation framework for the comparison of NIDS proposals. This approach also entails several important drawbacks, which will be discussed in detail in the following sections.

2.3 Data Collection Stage

The first phase of the KDD process involves the collection of the data with which network intrusions are detected. Network data[2] can be gathered at different locations.

Commonly, networks are organized in a hierarchical (tree) structure where the devices (typically, routers and switches) have visibility of the traffic entering and leaving the machines connected below them. Higher level (root or core) devices have a complete view of Internet-related traffic going to and coming from the computers (and network devices) located below their level. However, they have a limited view of horizontal communications (those between machines at the same level). The opposite is true for lower level (aggregation and access) devices [276]. Thus, the choice of the level at which to carry out the collection of network data has a bearing on the coverage of a NIDS and, consequently, on the volume of traffic to be analyzed and on the set of network attacks that can be detected.

Data collected from root network nodes corresponds to traffic coming from or going to a large number of network elements. As such, network data obtained at this level could allow Internet-related attacks to be detected by targeting as many network-connected machines as possible. However, some attacks would remain undetected because sometimes malicious activities go horizontally, and does not involve the root nodes. NIDS using root level captures need to process huge data volumes and must support extremely high throughput to avoid discarding traffic and to minimize detection delays. Also, due to the number of network elements they cover, they are easily affected by noise and perturbations, for example the connection of a new device on any segment of the network [134].

In contrast, the analysis of incoming and outgoing traffic of a small portion of the network, and captured from a lower level device, for example a datacenter switch, makes detectors more immune to noise and perturbations which occur at other network segments. Consequently, NIDS using this data require more modest throughput levels [134]. However, by design, their coverage is restricted to events occurring on a specific network segment.

Other scenarios are possible. Distributed collection of network data aims to provide a trade-off between network coverage, performance and stability. Partial captures and/or measurements may be carried out at several network points and gathered at a central location, overcoming the drawbacks of using any of the two previously described approaches individually [73].

Data collected by network probes to feed NIDS may have very different characteristics. Sometimes it consists of full copies of all data packets, such as those

[2] Throughout this chapter we use "network data" as a generic term to refer to the information provided to the NIDS to carry out its detection activities. A particular class of network data is "network traffic", with which we refer to a collection of captured packets with none or minimum processing. Note that, with this definition, "network data" may include also higher-level information including, for example, per-connection source/destination devices or average packet inter-arrival times.

captured by a network sniffer. Another option is feeding the NIDS with traffic summaries (measurements), normally in terms of per-flow data samples, at the cost of having a limited view of the traffic that is traversing the network. We elaborate on these concepts in the following sub-sections.

2.3.1 Capturing Raw Packet Data

Communication networks move data units, which are commonly known as packets, from one node to another. Every packet consists of protocol information arranged in layers which encapsulate the payload, i.e. the actual data being transferred. Each layer contains a set of headers (control information) that is appended to the information of the layers above and is required by network devices to be delivered to its destination [276][3]. This concept is known as encapsulation. Figure 2.2 shows the structure of a typical network packet and the encapsulation concept.

Protocol layers are necessary to move a message through the network. But not all the applications use the same protocol sets for this purpose. This means that not all the packets have the same headers with the same fields, and even equivalent fields for two different protocols may be located at different positions within the packet [276]. Also, the nature of the different fields that conform a network packet is heterogeneous, with text (user messages), numbers (lengths, window sizes), categorical information (such as control flags, IP addresses and port numbers), etc. This means that the size of network packets is also variable: from packets containing only control information necessary for protocol operation but without payload, to large application messages that need to be split into several sub-messages (due to limits imposed by some protocols) and are reassembled at the destination. All these factors complicate the creation of a uniform and structured dataset that will be needed to apply data mining methods.

Packet capturing tools (a.k.a. network sniffers) such as *Tcpdump* [144] or *Gulp* [260] can be used to retrieve and store raw network packets. They can run in a single computer, essentially capturing the packets: arriving to/departing from that machine (in-line mode), or from a network device configured to mirror all packets to the port to which the sniffer is connected (mirroring mode). In contrast to in-line mode, mirroring allows traffic from many machines to be captured but has a bearing on the performance of the device [134], which is affected by the increasing number of machines to be monitored and the growing speed of modern networks.

Capturing raw packets is a challenging task that requires high computational and storage requirements [55]. Insufficient bandwidth in the device/link used to carry out the captures, or in the machine in charge of storing/processing them, may cause packet losses [134], that may in turn reduce the detection abilities of the NIDS. In addition to that, if packets are ciphered using, for example, IPSec [96] (available for IPv4 and IPv6), access to the contents of the packets, including the headers of higher-level protocols, is impossible unless additional decryption mechanisms are

[3] Unless otherwise stated, throughout this document we assume that we are dealing with IP datagrams.

implemented. Access to raw packet captures, in contrast with pre-processed data, is of great interest as it enables the extraction of any kind of traffic-related feature that may enhance the detection ability of a NIDS.

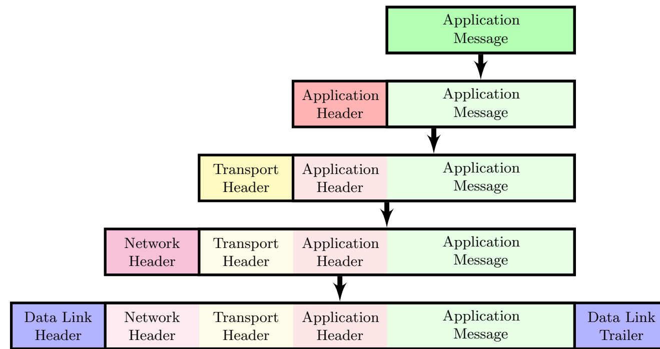


Fig. 2.2: Sketch of a network packet showing protocol encapsulation. Actual headers (fields, size, options) depend on the actual protocols used at the link layer (Ehternet/PPP/...), at the network layer (IPv4/IPv6), at the transport layer (TCP/UDP/ICMP/...), etc.

2.3.2 Capturing Flow-level Data

A *flow* is a collection of related packets that share a common set of attributes. These include network, transport or application header fields (IP addresses, IP protocol, TCP/UDP source ports, Type of Service...) as well as information related to the handling of packets on the device, such as physical input and output ports. Flows can be either unidirectional or bidirectional; in the later case they are also called *sessions* [65].

Flow-related features can be derived from raw packet data, as will be explained in Sections 2.4.1.2 and 2.4.1.3. However, the collection and reporting of flow-level data records is an increasingly common capability of network devices such as routers or switches. This feature aims to facilitate network management and supervision. Device manufacturers implement their own mechanism to gather and transfer this data, commonly adhering to the IPFIX standard [65], which was derived from Cisco's NetFlow v9 [64]. IPFIX-capable devices compute flow measurements in near real-time, without storing raw traffic, making use of sampling and accounting procedures [242].

Flow records computed in the device are exported to a *collector*, which stores them [65]. Each record contains the basic information about a flow, plus additional elements, including among others[4]: time stamps for flow start/finish times, num-

[4] A complete list of IPFIX elements can be found at `https://www.iana.org/assignments/ipfix/ipfix.xhtml`

ber of bytes/packets observed in the flow, the set of TCP flags observed over the life of the flow, etc. Note that the payload, the actual end-user application data interchanged by means of the packets, is an optional field of flow records and, when present, it normally includes just a few octets. Therefore, flow captures may be far more compact than raw packet captures. However, working with flow data adds latency that degrades real-time operation [242, 134].

Network flow data is useful to detect network attacks that leave footprints in traffic patterns such as traffic volume, timings of packets and/or communicating endpoints (sources and destinations of data: machines and/or ports). The usefulness of flow data for NIDS ultimately depends on the sampling mechanism used to make the flow measurements. Sampling methods take partial measures to obtain an approximation of the actual values when the number of connections is too high [83]. They reduce the impact on the performance of the device: only 1 out of n packets are checked, n being a manager-selected parameter. Accordingly, flow sampling parameters determine the behavior of the estimation process and play an important role in the extraction of flow records with useful (accurate) information for intrusion detection [199, 245]. For example, choosing a high sampling rate (meaning that a large number of packets per connection are checked) reduces the risk of obtaining poor estimates and the potential loss of vital information for NIDS [199], but there is a performance penalty.

2.4 Preprocessing: Derivation of Features

The preprocessing stage involves the derivation and transformation of feature vectors from the collected network data. As a result, a dataset of structured records, each of them composed of a set of explanatory features and usually, a response variable; is obtained.

2.4.1 Derivation of Explanatory Variables

Explanatory variables or features describe/represent the characteristics of a traffic observation (packet, connection, session). The derivation of explanatory variables or features is an important task of KDD because the detection capability of data mining methods highly depends on the information provided by the input variables. In this section, we discuss the different types of explanatory features, the procedures required to obtain them from network collected data and their usefulness for detecting network attacks.

Feature extraction can be done at different levels to enable network traffic analysis (methods to infer information or patterns from network-related data such as timings and counts of network packets) even when encryption is used [214]. We use the scheme proposed by Davis et al. [71] to describe those levels (using *flow* however, instead of *connection*-based aggregations) and the different classes of variables that are obtained at each level (see Figure 2.3).

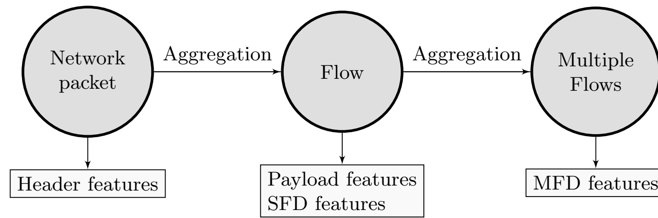


Fig. 2.3: Feature derivation process.

2.4.1.1 Packet Header Features

A raw packet can be converted into a feature vector by parsing the distinct set of header fields of the different protocols in use (from different layers of the TCP/IP stack). Packet header features are useful to detect protocol misbehavior and low level network attacks that aim to provoke protocol malfunction, such as spoofing (impersonation) attacks or attacks that do not leave their footprint in the application message [179].

Given the diversity in the structure of raw packets, and in the data types of the different protocol headers, the derivation of a common vector of features from this data [179] can be done in two different ways: (1) assigning missing values to those features that do not appear in a specific packet, or (2) removing fields that are not common to all protocols at the same layer. While the former leads to a more sparse representation of the data, the latter technique results in the loss of information that may be relevant for the detection of some attacks. In order to deal with both shortcomings, some authors proposed using different detectors: each one working with a particular application protocol (service) [173] or each one analyzing the headers of different protocol layers (for IP and TCP) of the protocol stack [198].

A main issue when working with packets (raw or converted into feature vectors) is not the preprocessing (parsing) procedure, which is straightforward, but the enormous throughput of packets sent to/received from a broadband link. As previously mentioned, capturing, preprocessing and storing the packets in real time is a challenging procedure of growing difficulty, which can be partially addressed through the use of different processing points working in parallel [55]. Also, encryption such as IPsec used at the network layer (either with IPv4 or IPv6) makes extracting higher-level header features a challenging procedure; only network and data link headers can be extracted easily without decryption mechanisms. In these scenarios, a per-packet analysis becomes difficult, but some forms of network traffic analysis (using flow-level data) are still feasible [274].

2.4.1.2 Payload Features

Payload features are those obtained from the content of network messages, that is, the data exchanged between applications. For NIDS, this content is commonly analyzed using techniques based on natural language processing [216], deriving vectors

with the frequencies of characters [173], words [198] or n-grams (all subsequences of n consecutive symbols) [114, 233, 19, 123] that appear in the payload.

Payload features are useful to detect network attacks that may exploit host vulnerabilities, such as injection or shell-code attacks (buffer overflows, privilege escalation, etc.). These attacks do not leave a specific footprint on packet headers, but the attack evidences can be found inside application-level messages [233, 19].

One of the main characteristics of payload features is their high dimensionality. Note that, in general, the feature vectors obtained from payloads are very long, and may require additional dimensionality reduction procedures (performed in posterior phases of the KDD process).

For connection-oriented applications, an application message may be split before transmission into a collection of packets; the number of packets depends on factors such as the message size or the maximum segment size parameter [276]. Therefore, processing methods aiming to extract meaningful features from application payload need to reassemble the packets to reconstruct the original message.

Processing application messages is also challenging in terms of computing capacity, memory and speed. The transmission of high volumes of data at high speed and involving connections between a large number of devices may stress not only the capture mechanisms, but also the feature extraction tasks. Enough capacity (processing speed, RAM, storage) must be guaranteed to properly carry out these tasks [274].

Encryption algorithms, that may be used at any protocol level to provide security (confidentiality, integrity, authentication), make the extraction of payload features even more difficult [221]. Efficient decryption mechanisms are essential in those scenarios if access to plaintext (non-encrypted) payload is a requirement. Such functionality typically requires additional processing infrastructure, such as TLS interception proxies that cut off encrypted traffic to decipher and inspect it before being forwarded to the actual server [252], or modifying the ciphering suites used for communication [182]. Drawbacks of these mechanisms lie in the increase of network latency and in the reduction of service security [84]. In practice, access of NIDS to the contents of encrypted messages is an inconvenient task, and the use of host-based intrusion detection systems (HIDS) may be more adequate as they have access to plaintext messages [54].

2.4.1.3 Single Flow Derived Features (SFD)

SFD features are those that correspond to a collection of packets sharing any property, i.e., a flow. Packets may be aggregated using different criteria until a given event, such as the end of a TCP connection or a timeout trigger, is detected [229]. Then SFD features are extracted to summarize different properties of this aggregation of packets, that can be either statistical measures or common header fields (from IP and transport layers). Features directly extracted from application messages (payload features) are not considered SFD features. As a result of this derivation, a mixture of nominal and numerical variables for each record (flow) is obtained [229]. Tools to compute SFD features from captures of raw packets include Argus

[191] and Bro [229]. However, the common approach to obtain SFD features is to use the flow-related information provided by IPFIX-capable network devices.

A few NIDS papers that make use of SFD features extracted from raw packets superficially describe how these features have been derived. Different header fields are used as aggregation keys, the most common ones being the source IP address or the source-destination IP address pair [59, 292, 82]. In [35] IP address pairs and TCP/UDP ports are used as aggregation keys, and those values are used as SFD features. Other header fields common to all packets in a flow, such as initial sequence numbers and TTL values, have been used as SFD features in [27, 212, 227], together with aggregated measurements such as: dropped and re-transmitted packets; average, maximum and minimum packet sizes; inter-arrival times; bit/packet rates from source to destination and vice-versa, etc. In [225], features derived from header fields are not considered valid, as they can be spoofed by attackers. They use the same aggregation keys, but instead they analyze average packet inter-arrival times and average packet lengths. A set of more complex SFD features, such as the coefficients from polynomial and FFT decomposition of inbound and outbound packet sizes, is derived in [136], to prove that their utilization makes a NIDS more resilient to some obfuscation attacks, explained at the end of this section.

SFD data sets are much smaller than raw data sets because payload is removed and common header features are summarized. This compactness is fundamental for many NIDS, since it enables the operation in near real-time in high-speed networks [274], although care must be taken when sampling is used, as vital information may be lost, see Section 2.3.2. Also, SFD features are useful for network traffic analysis even when encryption is used, without the need for additional decryption mechanisms [171].

SFD features enable the detection of network attacks that leave a footprint in the summarized statistics of a flow, e.g., affecting the volume, timings or the number of connections. These footprints also include the presence of connections in unexpected contexts, or connections with suspicious/uncommon header values (port numbers, source addresses), etc. [134]. Some works have proposed the use of SFD features to detect network attacks that are normally searched for inside the payload [312, 167]. However, the viability of this approach is questionable, given the high rate of false alarms they report.

Basic SFD features can be easily tampered with by intruders, by means of obfuscation techniques. These techniques have been proposed as a protection measure to shield network activities against network traffic analysis [278]. They include the insertion of artificial inter-packet delays, and the injection of malformed and spurious packets. Unfortunately, attackers can also use obfuscation, modifying harmless and attack traffic to make their SFD statistics indistinguishable, not allowing detection by the NIDS [238, 306].

2.4.1.4 Multiple Flow Derived Features (MFD)

MFD features are derived aggregating information pertaining to multiple flow records. Thus, they contain higher level statistics providing a more abstract insight

about the traffic traversing the network, and enabling additional traffic analysis capabilities. To compute MFD data, flows are commonly grouped using criteria such as "flows inside a time window T" or "last n flows". Then, sets of statistics are computed for each group [175]. Note that packets within a flow share many properties (SFD features), while flows aggregated in a group may not share any common characteristic beyond being in temporal and, sometimes, spacial proximity.

Normally, NIDS papers proposing the use of MFD features do not provide enough details about their derivation. The main aggregation criteria used are time windows and flows coming from/going to the same IP address (or to the same set of network prefixes in coarse grain scenarios). In [59, 82] those aggregations are used to extract MFD features such as counts of different ports and hosts, packets per second, ratios of packets with specific TCP flags (RST, SYN) set, etc. Entropy values that indicate the variability of discrete SFD features (such as source and destination IP addresses and ports) are MFD features used in [141, 175, 305]. Covariance matrices are obtained in [149, 319] to embody the distribution of continuous SFD features. In [179] MFD features are computed using a time window: for every connection, the number of past connections with similar SFD values for source and destination IP addresses, service and protocol flags are computed. Other authors [148] use a fixed-width window of flows as aggregation criterion to subsequently apply the Discrete Wavelet Transform [145] and use the coefficients of this transform as MFD features.

MFD features are useful for identifying attacks that spread their activities across multiple connections, such as flooding attacks (DoS attacks) or network scans [319, 59]. The success of attack detection using MFD features depends on the size of the window used to aggregate flows, as attackers can choose to spread their activities over the time to elude detection. Consequently, the use of different window sizes should be considered [179].

On the downside, when MFD features are used as the sole data source for the detector, it is difficult to trace the source of the attacks, due to the absence of packet or flow identification fields [175]. Furthermore, calculating these features has an additional cost in terms of processing, memory resources and time.

2.4.2 Class Labeling: Derivation of Response Variables

Response variables, or labels, are defined to explain the actual nature of a data record: non-malicious or malicious. In a fine-grained scenario, labels may even represent the specific attack type. These variables are usually required for evaluation purposes, to compare the output of a detector with the actual label. Also, the availability of labeled records is mandatory for training supervised data mining approaches such as classifiers (see Sections 2.7.1 and 2.8).

As labels are vital to properly construct and evaluate NIDS, the labeling process must be approached carefully to obtain a dataset without false positives or negatives. False negatives correspond to attack records that are labeled as normal traffic because they are not properly identified by the annotator. On the contrary, false

positives refer to normal traffic labeled erroneously as malicious. Such mislabeled data leads to detectors raising alarms for normal traffic while overlooking attacks.

There is no single way to add labels to dataset records. Some methods require human intervention, while others are automatic. Manual labeling is done by analyzing each record in the dataset, along with its context, and deciding whether a record is malicious or not. Ideally, this task should be carried out by expert annotators, because the identification of malicious patterns requires domain-specific knowledge, necessary to provide highly reliable labels. However, human experts are a limited and valuable resource that is not always available given the great effort required by this task. For this reason, manual labeling is often considered impractical [265].

Active learning techniques aim to reduce the effort required to manually label records. To do so, only the most informative samples, those providing non-redundant information, are taken as candidates to be manually labeled [12]. Relevancy of samples is computed leveraging on the output of a supervised model trained using data that is already labeled, e.g., based on class probabilities, distances, outcome of voting committees, etc. Active learning has proven to be useful to produce labeled datasets that may be of small size but result in detectors that perform better than others trained with larger datasets in which redundancy has not been removed. However, for evaluation purposes, the number of labeled records obtained by means of active learning may be insufficient.

To fully automate the labeling process, external tools can also be used. A NIDS detector specialized in matching signatures for known attacks (misuse detection) is used for labeling in [195], together with public databases of attacks and public lists of blacklisted hosts. The objective is to minimize the number of false negatives to which misuse detectors are prone [294]. Unsupervised labeling by means of anomaly detectors is another alternative used in [30, 93]. They focus on identifying deviations from normal (majority) traffic statistics, but detected events may contain false positives [271]. In [30] a clustering algorithm is used to differentiate normal vs. attack records: samples in small or sparse clusters are labeled as attacks, and the remaining ones as normal. Proposal [93] uses several statistical anomaly detectors, whose output is combined to reduce the risk of false positives. Note that in any case, mislabeled records may still remain in the dataset. In order to deal with these errors (false positives and negatives), some NIDS authors differentiate between "normal" and "background" traffic, understanding that background traffic may contain intrusions not detected by a misuse detector [195]. Other works associate different certainty levels to records labeled as (possible) attacks by an ensemble of unsupervised anomaly detectors [93].

Yet another way of generating labeled datasets is through a synthetic environment in which well identified network attacks are injected into a network at specific, and well controlled, times [265]. The injection schedule is used to do the labeling, by matching attack times with the time stamps of network records. Scheduled labeling ensures the identification of all malicious records while keeping the labeling cost under control. A disadvantage of these process is the lack of realism in the scenario, as both the normal and the malicious traffic is artificially generated.

Dataset	Raw captures	Payload features	SFD	MFD	Labeling method[1]	Duration	Year of Captures	Non-intermittent capture	Real scenario	#Articles
DARPA 1998 [158]	✓				MS	9 weeks	1998	YES	YES	6
DARPA 1999 [188]	✓				MS	5 weeks	1999	YES	YES	5
KDD-CUP 99[2] [179]		✓	✓	✓	MS	9 weeks [1]	1998	YES	YES	36
NSL-KDD[2] [284]		✓	✓	✓	MS	9 weeks [1]	1998	YES	YES	13
Kyoto2006+ [272]			✓		A	34 months	2006-2009	YES	YES	3
MAWILab [93]	✓				A	15min daily traces	2001-	NO	YES	3
ISCX-2012 [265]	✓	✓	✓		S	1 week	2010	YES	NO	3
TUIDS [42]	✓		✓	✓	S	1 week	2011	YES	NO	3
UNSW-NB15 [212]	✓	✓	✓	✓	A	31 hours	2015	NO	NO	2
UGR16 [195]			✓		AS	19 weeks (132days)	2016	NO	YES	-
NGIDS-DS [121]	✓				S	5 days	2016	YES	NO	-

[1] M=Manual labeling, A=Automatic labeling, S=Scheduled attacks
[2] Derivatives of DARPA datasets

Table 2.3: A summary of datasets available for the evaluation of intrusion detection systems.

Note that labels may be (1) unbalanced, because in a network there is more harmless traffic than attacks, and not all attacks are equally common; and (2) unrealistic when, in order to make it more balanced, the proportion of attacks is artificially high [302]. Both alternatives have a bearing on the performance of data mining algorithms [129]. Several authors modify the datasets in order to have a "fair" label representation for training their NIDS models [327, 236, 86, 218, 106, 163].

2.4.3 Publicly Available Datasets for NIDS Research

From the previous sections it can be seen that collecting and preprocessing realistic data for NIDS research and development are complex tasks, particularly in supervised scenarios where a ground truth (labeled records) is needed. Many NIDS researchers resort to publicly available datasets that help reduce this effort and also provide a common yardstick which is useful to compare against other proposals. However, their use presents some limitations and drawbacks, which we explore in the following paragraphs.

Table 2.3 shows a summary of popular NIDS datasets, the features available in each one, as well as the labeling method used by their creators. It can be seen that most datasets provide already preprocessed records, and only a few of them provide the raw captures from which higher-level (SFD, MFD) features were derived. This is a clear drawback and, ideally, all datasets should include the raw captures, allowing researchers to obtain those header, payload, SFD and MFD features that they find more adequate –instead of being limited by those already included in preprocessed data.

A deeper investigation about the generation process shows that some datasets [158, 188, 284, 272, 93, 195] are captured in real networks during the operation of real users. Most of these datasets, due to privacy restrictions, need to be anonymized without affecting the original characteristics of the data, which is an important challenge on its own [206]. Oftentimes, network managers are not willing to collaborate in dataset creation as there is a risk of exposing vulnerabilities in the networks they are operating (and, therefore, in their organizations). For this reason, synthetic network environments are the main alternative to obtain captures from network traffic.

Synthetically generated datasets [265, 42, 212, 121] contain traffic captured in synthetic environments, designed with the aim of being as realistic as possible, but with limited infrastructure; hence, these datasets are characterized by including a small number of hosts (sources/destinations of traffic) and limited traffic heterogeneity, which may be easy to model –resulting in biased and scenario-dependent detectors. Moreover, traffic generation methods include programs that try to mimic the operation of the real users [265, 42, 212, 121]. In general, the adequacy of synthetic datasets is in question as they do not truly reflect the operation of actual networks in terms of size and variability of user behavior (legitimate or malicious) [195].

Another issue of these datasets is the short duration of captures, commonly lasting only a few weeks or even days (see Table 2.3). Networks are changing environments where traffic patterns, harmless and anomalous, change throughout time and rarely manifest long stationary periods [271, 292]. Short duration captures over intermittent periods limit the correct evaluation of the adaptability of the models when the network changes. Some commonly used datasets were created more than a decade ago and, therefore, contain outdated traffic, meaning that old protocols and services are present, while current ones are not. Also, ciphered traffic is omitted. In summary, the included traffic patterns do not represent current networks. At any rate, being recent does not guarantee being representative of current networks, especially if the dataset is synthetic [195].

The methods used to label records are also an important factor to take into account when choosing a public dataset (see Table 2.3 and Section 2.4.2). Manual labeling is the most accurate method [265]. Synthetically generated datasets provide good quality gold standards due to the way in which attacks are inserted but, as we mentioned, they may not reflect realistic patterns. Labeling traffic captured from real data is problematic, as normally it is performed using automated tools that do not guarantee the lack of false positives and false negatives. These errors may have a significant effect on the effectiveness of a NIDS proposal [271].

The most used datasets in the literature (see Table 2.3) are the DARPA datasets and their derivatives (KDD-CUP 99 and NSL-KDD). Such databases have been largely criticized along the years due to statistical issues [203, 197, 284, 51]. Despite better alternatives have become available in the last few years, the DARPA datasets and their derivatives are still used to develop and test intrusion detectors [268]. Instead, more recent datasets have not received enough attention by NIDS authors.

In summary, using publicly available datasets, as most NIDS researchers do, may be convenient, but they are not free of issues of which researchers must be aware. These datasets are not perfect, given the difficulties to emulate a realistic networking scenario or to share data from a real corporate network. They can be very useful in the initial phases of NIDS research (thus, the efforts of dataset authors must be appraised), but more work needs to be done in dataset creation and evaluation. The traffic that makes up the datasets must be as realistic as possible and associated labels should be of good quality, to give validity to the evaluations of NIDS proposals.

2.5 Preprocessing: Data Cleaning and Feature Transformation

The main characteristic of the feature vectors (records, samples) obtained from the previous stage is their heterogeneity. They include discrete (categorical and numerical) and continuous (numerical) features. As some data mining methods cannot deal with this heterogeneity, it is common to transform the features to obtain homogeneous records (all numerical or all categorical). Also, data mining algorithms are very sensitive to noise in the data caused either by outliers or errors during the data capturing process. The procedures used during this second preprocessing stage involve the derivation of new features, the modification of existing ones and the elimination of noisy samples, all in order to build a "clean" dataset [234].

Unfortunately, little attention has been paid to this stage by NIDS authors. Most papers do not provide enough details about the exact cleaning and transformation procedures they perform, making their analysis difficult and hindering the reproducibility [181, 45, 40, 140, 250, 5, 259, 186, 266, 227].

2.5.1 Noise Reduction

Network data may be susceptible to errors during processing. Such errors, commonly denominated noise, are reflected in different forms (missing values, outliers...). Noisy data affects the learning capacity of data mining detectors negatively, making them prone to failures [336]. Thus, noise reduction procedures aim to increase the effectiveness and accuracy of data mining algorithms. Noise can be reduced by deleting those samples showing higher distortions with respect to the majority of elements in the dataset, either in a single feature with extreme or missing values, or in several features. Note that, as we describe in the remainder of this section, other transformation methods also contribute to the reduction of noise in an indirect manner.

Only a few NIDS papers explicitly describe the noise reduction techniques they use. Examples are [267] and [82], where outlier removal is done before feeding the dataset to their respective detectors. The former method uses the Mahalanobis distance to rank samples and remove the 0.5% with the largest values, whereas the latter technique uses an interval for every feature. To do so, limits away from the mean value are calculated taking 10% of the distance between the maximum and minimum feature values.

2.5.2 Transformation from Categorical to Numerical

There are different approaches to convert a symbolic feature into a numerical representation. A naive conversion is called *number encoding*. It keeps the dimensionality of the dataset, replacing with a number each possible categorical value. This method has been used to simplify a probabilistic classifier for NIDS in [34] and to cope with the incompatibility of categorical data of the model in [14].

One-hot encoding adds a new binary feature (dummy variable) for each categorical value in the variable, and assigns a positive value (1) to those observations sharing the category represented by the binary feature, and a zero value (0) otherwise. This approach has been commonly used in NIDS proposals to enable the use of distance calculations [236, 159, 72, 118] without the need to use a heterogeneous measure. Other NIDS, whose models cannot work with categorical values, also use this transformation [320, 264, 74, 148, 220, 192].

A variant of the previous method is called *frequency encoding*. Each feature value receives a weight according to the frequency of appearance of that value in the data. This method was firstly proposed for NIDS in [86], because it is useful to work with distances. Based on this first approach, frequency encoding was also used later in [106].

In summary, transforming features is a common task in NIDS to deal with the heterogeneity of network data (IP addresses, service types, protocol flags, timings...). Number encoding maintains feature dimensionality, and is useful to simplify value comparison and to reduce memory utilization. However, while distance or order relationships are inherent to numerical data, this is not necessarily true for categories encoded numerically. Therefore, this solution may lead to erroneous conclusions when using certain data mining algorithms, e.g., those using distances. One-hot and frequency encodings work well with data mining methods that make use of the arithmetical properties of features, at the cost of a significant increment of data dimensionality [2]. For example, in the context of a large corporate network, where features with the source and destination of connections may take thousands of different IP address (categorical) values, this transformation results in an extremely wide vector of binary/frequency (numerical) features. Such sparse data slows down and damages the performance of data mining algorithms [128]. Another drawback, common to one-hot and frequency encoding methods, is that they assume that all the values of the categorical variable are known in advance, something that may not be true in changing environments such as networks (communications may reveal new IP addresses). In this situation, an encoding in use may become invalid.

2.5.3 Feature Scaling

Feature scaling, also called feature normalization, is used to limit the range of values of a numerical variable. It has been demonstrated that scaling procedures improve the convergence time of optimization algorithms, as they narrow the solution search space without negatively affecting the accuracy [178, 304]. Applying scaling methods may also reduce the skewness in the data as well as the bias caused to some data mining algorithms [285, 267].

The use of feature scaling has been documented in several NIDS articles. In [175], mean normalization is carried out to scale numerical values before the application of principal component analysis (PCA, see Section 2.6.1.3). The method uses the mean and the difference between maximum and minimum values to center the data. A logarithmic transformation (replacing a value with its logarithm) is used in [264, 40]

to reduce the variability of features with wide value ranges. Min-max normalization uses the minimum and maximum values of a variable to adjust its range to values between 0 and 1; this method is applied in [15, 14] and [264, 320, 192] to reduce the training time of the models used for detection. The same approach is used in [218, 74, 82] to avoid the bias caused by features with wide value ranges in distance-based algorithms. Standardization takes the sample mean and the standard deviation to transform a feature to have zero mean and unit variance; it is used in [236, 86, 283, 72, 118] to improve, again, the behavior of distance-based algorithms. Both standardization and min-max normalization methods are used in [13] to improve the performance of an optimization algorithm to estimate the parameters of their detection model.

Most data mining algorithms, excluding tree-based algorithms, are sensitive to the presence of features with different scales in the data. When data is unequally scaled, algorithms give more weight to variables with higher values, resulting in biased models [324]. Accordingly, the use of methods to re-scale features is often mandatory. The suitability of a scaling method depends on the characteristics of the data mining algorithm and the data itself. Standarization requires knowing the mean and variance of a feature beforehand, and it is recommended for algorithms that assume that data is normally distributed [324]. If this assumption is not made, other scaling procedures may be more appropriate. Min-max and mean normalization re-arrange feature values into smaller intervals; again, a priori knowledge of feature ranges is required. Additionally, for features with most values within a very narrow range, feature scaling methods result in a reduction of the differences between (re-scaled) values when outliers are present, disturbing the operation of some data mining algorithms [74]. Logarithmic scaling skews the data towards large outliers, while standardization, mean and min-max normalization lessen the significance of the inliers [2]. Noise removal, as well as the discretization methods that will be explored in the following section, may help in reducing these side-effects.

The need for previous knowledge about the value ranges of features is a major obstacle in dynamic scenarios such as streaming contexts, as this information may be unknown. Methods to re-estimate feature statistics as new samples arrive may be necessary. Logarithmic scaling is also useful in this context as it does not require any parameter.

2.5.4 Feature Discretization

Discretization is used to transform a continuous feature into a discrete one [189]. Although the use of discretization techniques is largely discussed in the machine learning literature [104], it is not that pervasive in the NIDS literature. Discretization works by selecting split or cut points over the continuous space of a variable to create subdivisions (intervals) of this space. Then, each continuous value is replaced by the corresponding interval identifier.

Variables are discretized with the purpose of increasing the learning speed of some detection algorithms, such as those based on induction rules [81]. In decision trees, the use of discretization improves the effectiveness of the learning procedure,

resulting in smaller and more accurate trees [241]. Discretization also reduces the complexity of algorithms based on the Bayes theory (by using summation instead of integration terms), while increasing the accuracy of resulting classifiers [189]. Furthermore, it is useful to remove noise from data and to reduce the side-effects of outliers [74], at the cost of some information loss.

The simplest discretization methods are Equal-Width (EWD) and Equal-Frequency discretization (EFD) [104]. EWD splits the original feature range into k intervals of equal width; EFD instead creates as many intervals as needed, in such a way that each interval contains the same number n of samples. Consequently, higher values of n will result in a smaller number k of intervals. A similar method uses K-means clustering, which breaks the range of features into k intervals and replaces the original value of a feature with the centroid of its corresponding range [74]. In all cases, the user must select the parameter that affects the number k of intervals.

Other popular, more complex discretization methods used for NIDS that set the number of intervals automatically are Proportional k-interval discretization (PKID) and Entropy Minimization Discretization (EMD). PKID selects a number k of intervals in such a way that each interval contains a number of samples that is equal to k, trying to find a trade-off between the choice of k and the sparseness of the data [315]. Feature intervals are selected as a function of the conditional probability distribution of the data. As a result of PKID, data is distributed proportionally among the bins, and the resulting discretization is adequate for naive Bayes classifiers [316]. EMD is based on the Minimum Description Length (MDL) principle, which seeks the minimum number of intervals with maximum information [90]. It recursively selects as interval cut points those elements with minimum values of information entropy with respect to the class, i.e., less representative areas of the continuous space of the feature. The intervals created are dominated by a single class [315]. ML algorithms based on decision trees include embedded discretization methods similar to EMD: a continuous variable space is split by selecting the cut point that yields the maximum information gain from the split [241].

Another discretization method, suitable for very dynamic scenarios such as networks, is proposed for NIDS in [74]. In such scenarios, feature ranges may be unknown in advance and applying most of the previous methods is impractical. For dealing with this issue at training time, each dynamic-range feature is clustered into k groups applying K-means. After that, k Gaussian distributions are obtained (one for each group). Next, as new samples arrive, each dynamic range feature is replaced by k new features, obtained using the Gaussian distributions and containing the probability of the original feature value pertaining to each group.

The benefits of using discretization methods for NIDS have been documented in many studies [49, 166]. The main shortcoming of simple methods is the need to choose the number k of intervals (or the per-interval frequency n for EFD). This is a non-trivial process, as a small k entails the removal of useful information and a large k results in the inclusion of noise [2]. Methods to automatically select the number of intervals based on the class information, such as PKID and EMD can be used, although the computational cost will increase. A common drawback of most

discretization approaches is that they do not work properly when processing data on-the-fly, because neither feature ranges nor labels (for those methods requiring them, such as EMD and PKID) are known a priori. Previously unseen values would invalidate the parameters of the methods. These issues may be tackled using self-adaptive, online discretization methods instead [99, 244], but no reference to them has been found in the reviewed NIDS literature.

2.6 Data Reduction

In the previous sections we have discussed the derivation of an exhaustive set of features, which later will be used to build the detector. This dataset may be huge in terms of the number of samples, and also in terms of the number of features per sample. In this context, it has been shown that using the full feature set extracted from traffic is ineffective for intrusion detection, due to the inclusion of irrelevant variables that contain redundancies and hinder the learning capacity of data mining algorithms [85].

In this section we discuss methods to reduce the dimensionality of the dataset with the aim of: (1) reducing the computational cost and (2) improving the accuracy of intrusion detectors. We describe some techniques that reduce the number of variables in the feature set, as well as others that reduce the number of samples (records) in the dataset.

2.6.1 Feature Dimensionality Reduction

Feature dimensionality reduction procedures try to cut down the number of features considered by the NIDS by discarding useless information or reducing redundancy. There are two main alternatives to do this: removing explanatory variables, or projecting the original explanatory variables onto a lower dimensional space.

2.6.1.1 Manual Feature Removal

The simplest feature reduction method is the manual removal of "irrelevant" features. It requires a profound expert knowledge of the domain, as those features are excluded from the next KDD phases [140].

As previously mentioned, many data mining algorithms cannot work with some types of attributes (categorical, numerical). To deal with this issue, many authors simply opt to remove features instead of applying preprocessing methods (see Section 2.5). For example, in [267, 283, 137, 120, 16, 94] all categorical variables are removed because their data mining detectors are unable to deal with non-numerical data. Note that this removal may result in important information loss [137].

Manual exclusion of features based on other criteria has also been described in some NIDS papers. In [149, 319] only MFD statistics of the dataset are used for DoS detection, discarding other features. In other proposals, constant features

are removed as they do not provide useful information [106, 27]. In [294] features such as IP addresses of attacking machines and attacking times are removed, as they reflect information clearly related with the simulated environment where the dataset was obtained from.

2.6.1.2 Feature Subset Selection (FSS)

These techniques select a subset of the whole feature set that is assumed to be the most relevant to solve the problem at hand. FSS methods reduce the cost of data mining models, help to prevent model over-fitting, and enable a better understanding of the data because redundancies are removed [254]. FSS methods can be classified into three main categories: wrapper, filter and embedded methods.

Wrapper FSS approaches measure the benefit that the inclusion of a given feature (or set of features) has over the performance of a predictive model. This can be seen as an optimization procedure that looks for the feature subset that maximizes the predictive capacity of the model. As such, and in order to measure the quality of the predictive model, they require the data to be labeled [2]. Different methods to select feature subsets have been described for NIDS, including sequential greedy methods [123] that add the feature that improves the model accuracy in a greater degree in each step until there is no a remarkable improvement; and methods based on meta-heuristics that guide the search of a solution in the space of subsets of variables. Examples of meta-heuristic FSS algorithms used in NIDS include Genetic Algorithms [161, 294], Firefly Algorithms [262] and Simulated Annealing [185].

In contrast to wrapper methods, results of *filter FSS* methods do not depend on the predictive model used in the next KDD phase. The relevance of features is computed through an evaluation of their characteristics using correlation, mutual information, consistency, variance or similar metrics [2]. Normally, filter FSS methods are computationally cheaper than wrapper methods, and more appropriate for datasets containing unlabeled data or a large number of features [254].

Correlation-based Feature Selection (CFS) methods [122] choose those features that are highly correlated with the response variable (the label) while showing a low correlation with the remaining features. To compute such correlation, Pearson's correlation coefficient is commonly used, although this coefficient is only able to express linear relations [2]. Other correlation-based filter methods use Mutual Information (MI) (based on entropy), which is able to measure complex relations [68], but can be applied to discrete features only. CFS methods based on Mutual Information have been used for NIDS by selecting the features that maximize feature-class mutual information [40] and include features that maximize feature-class mutual information and minimize the redundancy with respect to other features [15, 14].

Consistency-based methods evaluate the constancy of feature values with respect to class labels [70]. As such, features with stable values in samples belonging to the same class are selected. For NIDS this method has been used in [49, 166] in conjunction with two CFS methods, one based on Pearson's correlation coefficient and the other one based on MI [332].

Analysis of Variance (ANOVA) tests [77] decompose the variance of features among linear components, which can be used to compute the variance of a feature with respect to others. Then, redundant features containing similar characteristics to others can be discarded. For NIDS, this approach has been used to select those features that provide non-redundant information in [148].

Finally, *embedded FSS* approaches are either filter or wrapper methods integrated into prediction algorithms as part of the training phase. They are dependent on the particular data mining algorithm used, usually trees, as each training step comprises both ranking relevant explanatory variables and generating a model using the top-ranked ones [254]. For example, C4.5 [239] and C5.0 [241, 240] decision tree builders are used in [230, 218, 294] and in [149], respectively, to generate intrusion detectors. Both algorithms have a built-in filter FSS method that weights features using their Gain Ratio (an improvement over MI used to avoid bias towards variables with higher cardinality). Other authors use Random Forests for intrusion detection [326, 327, 293], which includes a wrapper FSS as part of the learning process. The feature relevance metric is based on the misclassification rate produced by the permutation of the values of the input variables [50].

2.6.1.3 Feature Projection

Feature projection techniques do not select a subset of the original feature set. Instead, they are used to obtain a projection of the features into a lower dimensionality space. It is a form of compression, based on the use of statistical and mathematical functions that find linear and non-linear variable combinations that retain most of the information of the original feature vectors [324].

Principal Component Analysis (PCA) [128] is one of the most common dimensionality reduction procedures. PCA applies a linear transformation to the original explanatory variables to map them into a space of linearly uncorrelated components. Dimensionality reduction is performed by selecting a number of components smaller than the original number of features [150]. Ideally, data provided to PCA must be scaled and free of outliers to avoid giving more weight to features with wider value ranges [267]. The main shortcoming of PCA is its inability to capture complex, non-linear relations between features. Several NIDS works use PCA to reduce the feature dimensionality of their data before training their models. In [257] several training and testing executions are performed using PCA in order to select the number of components that result in the highest performance. Proposal [72] selects those components that best discriminate between normality and attack, as indicated by Fisher's Discriminant Ratio.

In contrast to PCA, where the extracted components are orthogonal and ordered by the variance they retain, Independent Component Analysis (ICA) separates a multivariate signal into a set of statistically independent non-Gaussian components or factors of equal importance [143]. Another difference with respect to PCA is that ICA is able to capture non-linear correlations [128]. In NIDS, this method is used in [225] to obtain a representation of the data in which only those independent components that better characterize the malicious class are kept.

Auto-encoders, a type of Neural Networks (NN), have also been proposed for feature projection. They try to reproduce the input values (the original features) at the output layer. Dimensionality reduction is achieved by projecting the input onto a smaller space in the intermediate layers. Auto-encoders can model complex relations in the input data, not limited to linear relations [110]. Different auto-encoder methodologies have been proposed in the NIDS literature, including symmetric [107] and non-symmetric [266] NN architectures, sparse auto-encoders[5] [227], and auto-encoders based on self-taught learning techniques [243]. Four different auto-encoder models are used in [220] to determine the best NN-based projection method for NIDS.

Finally, different feature dimensionality reduction methods can be combined, either by applying projection methods to a subset of the original features obtained by means of FSS, or viceversa. For NIDS, PCA is applied to obtain a reduced projection of the features selected by a filter FSS method based on information gain in [257]. This combination is carried out in order to get a very compact representation of the data at the cost of increased complexity.

The two main challenges of feature dimensionality reduction methods are: (1) the choice of an adequate number of features, small enough to remove redundancies (and speed up the execution of data mining methods) but without harming the detection performance; and (2) dealing with *concept drift*, that is, adapting to unforeseen changes in the characteristics of input data, something common in networking scenarios [337]. Auto-encoders and FSS wrapper approaches require several costly training and evaluation steps with different dimensions (feature subsets in case of FSS) to find the most adequate size of the feature set [110, 2]. This does not happen with other projection methods (PCA, ICA) and most filter FSS approaches, as the inclusion or exclusion of features is decided by setting a threshold over their quality. The difficulty here is in tuning the threshold to determine an appropriate number of dimensions without discarding valuable information [128]. The availability of labeled data is also important when selecting a method to reduce feature dimensionality. For wrapper methods labels are compulsory. For projection and most filter FSS methods, the set of selected features depends only on the values of the features themselves, and therefore labels are unnecessary [2]. Data projection methods are widely used as they uncover implicit relations between features and may reach better reduction ratios than other methods [2]. However, because projections do not contain the original values extracted from traffic samples, their main shortcoming lies in the lack of explainability of detected attacks.

2.6.2 Sample Dimensionality Reduction

The cost of training most data mining algorithms depends heavily on the number of samples used to train the models. This may be a critical issue in scenarios where learning from data is applied under strict time constraints [114]. Techniques that

[5] These projection methods do not result in a reduced projection, instead they produce a sparse representation where the projected data is larger than the original feature set.

reduce sample dimensionality are sometimes applied with the aim of accelerating the learning step of those algorithms, either by removing duplicate or too similar samples, or by selecting representative samples that are used instead of the whole dataset[6].

In the context of NIDS, proposal [159] uses a tree-based clustering method to reduce the number of samples used to train the model. Instead of training with all the instances, the cluster centroids are used as representatives, thus achieving a significant reduction in the sample dimensionality. A similar method is used in [288, 137] to reduce the cost of performing the distance calculations required by their ML detectors.

2.7 Data Mining Stage

Data mining is the core stage of the KDD process, where the preprocessed and reduced data is fed into different algorithms whose objective is to identify patterns in the data and, ultimately, detect attacks. Due to its core role, most NIDS surveys (such as [41, 53]) focus almost exclusively on this stage. In fact, the authors of those surveys categorize NIDS proposals using the type of data mining algorithm used as the main criterion. In this survey we discuss this stage using a slightly different viewpoint, and present a novel taxonomy of NIDS methods based on two criteria: the detection goal and the learning approach.

The first criterion categorizes NIDS methods into three types (misuse-based, anomaly-based and hybrid systems) based on the meaning they assign to the concept of attack and the goal of the detector. The second criterion classifies NIDS proposals according to how they train or build the detector: in batch mode or incrementally.

2.7.1 Misuse-based, Anomaly-based and Hybrid Detectors

This nomenclature (misuse, anomaly) has already been used in previous surveys such as [53, 211] and, even if the definitions of these terms given in the literature are not always identical, the different versions do have some common grounds. In general, most authors have defined misuse-based systems as NIDS designed to detect well-known attacks using either a knowledge-base of attacks [274], rules written by domain experts [71], attack-signatures [28, 105, 53, 211] or attack-patterns [292, 179]. The authors do not specify the exact meaning of *signature*, *rule*, *pattern* or *knowledge-base*, but clearly some definitions refer to very rigid systems (signature, rules, expert knowledge), while others describe misuse-based NIDS as more flexible systems (patterns). Moreover, in any of these works, it is not clearly stated how this kind of identifying information can be extracted from data.

In the case of anomaly-based detectors, the definitions are vaguer and, oftentimes, with slight differences. These NIDS have been defined as systems whose

[6] These techniques are not to be confused with class balancing, whose aim is to reach a proportional, and maybe unrealistic, distribution of samples between the classes.

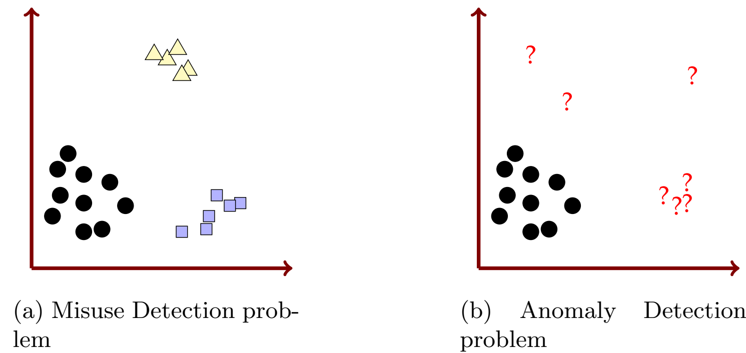
objective is to detect *exceptional* patterns [41], patterns that *deviate from other observations* [6], patterns that *deviate from the normal* [41, 53, 105, 179, 211, 71, 274] or patterns that *deviate from the expected* traffic [271], and also as systems *with knowledge of normal behavior* [28]. In this line, in many of the mentioned surveys, their authors state that an anomaly-based NIDS attempts to obtain profiles of the normal traffic, in order to compare the new incoming data with those profiles, identifying observations that deviate significantly from normality.

Often, different surveys classify the same NIDS differently, as evidence of the absence of a clear-cut categorization. For example, the classifications made for [19] and [327] in two widely referenced NIDS surveys are incompatible. In [41], Bhuyan et al. define the work by Ariu et al. [19] as anomaly-based, and the work by Zhang et al. [327] is categorized as a hybrid approach. However, it their review, Buczak and Guven [53] state that both proposals are misuse-based methods. In recent surveys, the inconsistency of these definitions has lead to the miscategorization of NIDS that use some data mining methods, such as supervised classifiers [41, 211, 160], identifying them as anomaly-based approaches although they present limitations to detect new and unknown classes of traffic [292, 60].

To avoid further misclassifications, we start laying the foundations for these definitions, providing more specific and up-to-date descriptions of misuse and anomaly based approaches. We focus on three aspects that depart from ideas and concepts of the data mining area, particularly from the area of anomaly/outlier/rare event detection [133, 235, 58] (see Table 2.4). Any combination of the characteristics shown in this table will result in a hybrid detector.

- **Misuse-based detectors** understand attacks and normality as categories/-classes within the data (see Figure 2.4a). They depart from the assumption that an attack is a minority category with specific patterns/characteristics that can be learned using the training data. Misuse-based systems require fully labeled data (all attacks must be correctly identified) at training time. Once a misuse-based system is built, the response obtained given a new observation is its correspondent or potential class according to the learned characteristics. Therefore, detection ability is limited to the set of attacks and/or normality classes modeled.
- **Anomaly-based detectors** assume that attacks can form categories with specific characteristics, but they can also be isolated observations (outliers) (see Figure 2.4b). With this in mind, these detectors regard that an attack is anything deviating from normal traffic and, therefore, they focus on modeling normality. Anomaly-based NIDS do not require attacks to be explicitly identified in the training data, and the output they produce during operation is vaguer than in misuse systems, since the returned information only states that an observation deviates sufficiently from normal traffic.

This categorization of NIDS is closely related to the degree of supervision required by the data mining algorithm applied. Classical supervised classification algorithms depart from a training dataset in which the instances are labeled as attack (or type of attack) or normal traffic. Their goal is to learn a predictive model



(a) Misuse Detection problem

(b) Anomaly Detection problem

Fig. 2.4: Visual description of the misuse and anomaly detection problems. In misuse systems, both the normal data (black circles) and the attacks (yellow triangles and purple squares) have specific characteristics and, thus, form classes. In anomaly detection scenarios, normal observations (black circles) are assumed to follow characteristic patterns that constitute well-defined classes, while abnormal, maybe malicious, patterns (red "?" symbols) may or may not have enough entity to form a class.

	Do the attacks form classes?	**What is modeled?**	**What is the response obtained from the NIDS?**
Misuse	Yes	Attacks and Normality	Normal observation vs. attack (or type of attack)
Anomaly	Not necessarily	Mainly normality	Normal observation vs. Abnormal observation

Table 2.4: Differences between misuse and anomaly detection from a data mining perspective.

from this data, which is then used to identify attacks in new incoming traffic [53]. As such, they are typically associated with misuse-based systems. On the contrary, unsupervised learning algorithms depart from a dataset of instances that are not labeled at all. Their goal is to identify the natural groups that underlie in the data [200] and, thus, are typically more related to anomaly-based systems.

In anomaly-based NIDS we can also find different types of weak supervision [131], where some kind of uncertainty is included in the labeling. To the best of our knowledge, two types of weakly supervised learning approaches have been applied to NIDS: semi-supervised learning and one-class classification. In semi-supervised learning [61], the data is partially labeled. Commonly, in NIDS, these methods learn in two steps. In a first step labels are ignored and data is clustered using unsupervised techniques. Afterwards, in the second step, all the instances of each cluster are labeled identically by leveraging the available labels, and assigning, for example, the label that appears the most. In one-class classification [286] instead, only some of the instances of one of the classes in the data are labeled. This serves to accurately differentiate between the elements of the labeled class with respect to those from other classes. In general, weakly supervised algorithms do not require as many labeled instances (which are difficult and costly to obtain, see Section 2.4.2) as

fully supervised approaches: one-class approaches only require the largest possible number of labeled instances of the class of preference, i.e., the one which we are interested to learn from (for anomaly-based NIDS, the normal traffic). However, semi-supervised methods work well even if there are only a few labeled instances of each class in the data [131]. Also, weakly supervised methods have better accuracy than totally unsupervised methods [131].

In the following subsections we describe NIDS works from each of these categories, commenting on the advantages and disadvantages of each approach. We also discuss the degree of supervision required, or typically assumed, by the different detection methods.

2.7.1.1 Misuse-based NIDS

As mentioned above, misuse-based systems normally leverage classic supervised classification methods. These algorithms have evolved and improved substantially in the past few years and, consequently, they are able to yield very reliable predictions when fed with enough, good quality data. The main advantage of misuse-based systems is that they are able to accurately model the patterns and characteristics of well-known attacks present in the training set, obtaining low rates of false alarms (false positives). However, most of these techniques have important limitations when dealing with new (zero-day) or mutated attack patterns [294], because they are not designed to identify new classes. Thus, they require continuous updates to extract new attack patterns, making them hard to maintain. Also, remember that this type of algorithms require a labeled training set, which is costly and difficult to obtain in real scenarios [2] (see Section 2.4.2).

The main difference between the various misuse-based systems proposed in the literature is the specific supervised classification algorithm used. Neural Networks (NN) are rather popular and, in NIDS, there are approaches ranging from the simplest network, the Multilayer Perceptron (MLP), [213, 148, 227], to complex Extreme Learning Machines (ELM) [27], Convolutional Neural Networks [192] and Recurrent Neural Networks (RNN) [264, 163, 320].

Other less frequent approaches include Support Vector Machines (SVM), based on finding the maximum margin hyperplane that separates the classes, [15, 185, 140, 148, 14, 123], k-Nearest Neighbor (kNN) approaches, based on distance calculations [86, 186, 118], the Naive Bayes algorithm, based on probabilistic theories [35, 49, 166, 185, 148, 294] and Decision Trees, based on constructing tree-like structures that output decision rules [149, 49, 185, 225, 162, 294, 218, 262]. As a more rare approach, a supervised probabilistic variant of the Self-Organizing Maps (SOM) algorithm [168] (which is an unsupervised clustering algorithm), has been applied to NIDS in [259] and [72], assigning to each cluster the most frequent label in its allocated records during the training phase.

The combination of supervised classification methods to form ensemble models has also been proposed for NIDS, since it reduces bias and improves classification accuracy [50, 263] compared to using only one model. The most common ensemble method is the Random Forest [327, 293, 292, 266], which is based on combining

many decision trees. However, other heterogeneous ensembles have also been proposed: combinations of Decision Trees and SVMs [230]; or even combinations of SVM, kNN and MLP classifiers [257].

We already mentioned that misuse approaches are only able to recognize what is learnt during training. In this context, in order to provide some flexibility and to avoid overfitting to the training data, some supervised methods include regularization or generalization terms [128]. An alternative approach to provide more flexibility to these methods and attempt to generalize better is proposed in [137]. This misuse-based detector is a variant of kNN, a distance based classifier, that incorporates a genetic approach. This genetic component makes every new sample evolve to perform its classification using a clustered representation of the training data.

We have seen many distinct methods that are used for misuse detection. A comparative analysis of them is not easy due to the lack of a common framework. However, some general conclusions can be extracted through an analysis of the algorithms applied. Among the simplest approaches we can find DT and Bayesian methods, which involve a straightforward learning mechanism. In contrast, NNs and SVMs need a thorough parameter tuning [128]. Most algorithms are costly to train and perform better when they are fed with large datasets. kNN is costly in the detection phase because, to obtain the label for a new sample, it requires the computation of the distance between that sample and the training set [2]. SVMs are good choices for learning from wide datasets (those with few records and many features). Regarding model interpretability, DTs are the most explainable, while the opposite holds true for NNs [128].

2.7.1.2 Anomaly-based NIDS

These detectors assume that an attack is anything that deviates from normal traffic, and thus, focus mainly on modeling "normality". Deviation of new observations from this normality has been measured in many different ways. As previously mentioned, while misuse-based detectors are commonly based on supervised learning schemes, we can find supervised, weakly supervised and non-supervised approaches for anomaly detection.

A main advantage of anomaly detectors is that they should be able to identify unknown attacks (mutated or zero-day attacks). Labeled data is not a requirement for such systems because normal profiles can be modelled using unlabeled or partially labeled network datasets, which are expected to have a low proportion of attacks [267]. A main drawback of these detectors is that, when deployed in real scenarios, they usually produce a high number of false positives, often related to noise or events that are not attacks (e.g., the connection of a new server in the network). Therefore, anomaly-based NIDS typically require companion methods to explain/categorize the alarms [109].

An anomaly-based NIDS identifies those observations that are very different from the normal majority traffic as attacks. As such, a threshold must be defined to determine if the level of deviation is enough to trigger an alarm. Anomaly-based

NIDS proposals differ in the data mining algorithm chosen, as well as in the way deviation is measured. Let us focus first on unsupervised approaches.

The first and most simple unsupervised anomaly-based systems are based on the extraction of frequency profiles from the patterns observed during training. These detectors assume that attacks leave as footprints patterns that do not appear as frequently as normal patterns. For example, in [236, 173, 198, 223, 312, 170] the authors flag infrequent values in traffic features as malicious. More sophisticated is the approach followed in [120]. Traffic of similar characteristics is first clustered, and the frequency of transitions between clusters is analyzed. Attacks are identified when uncommon transition patterns appear, leveraging thus on the temporal relationships among different types of traffic (clusters).

Other approaches are not based on frequencies, but on modeling the correlation structure of the normal data and assuming that attacks follow a different correlation structure. Two main approaches can be identified here: using correlation features directly, or using subspace representations of the correlation features. Examples of the first approach can be seen in [319] and [283]. In the first case the mean of the covariance matrices of normal data is computed. Later, records that deviate from this profile beyond a threshold are flagged as attacks. The second proposal computes the area of the triangle formed by the projection of any pair of variables in the Euclidean 2D space, which is used as an indicator of their correlation. The system generates a normality model by computing the Mahalanobis distance between all normal triangle area records, to obtain the parameters of a univariate Normal distribution. After that, a threshold for admissible normality consisting of n times the standard deviation of the normality profile is set.

Subspace representations of the correlation features by means of projection methods (PCA) have also been used for anomaly detection. For example, in [267] the authors use PCA to extract the major and minor components of normal data. The k major components are those capturing most of the variance. Incoming records with normal correlation structure and extreme values are flagged as anomalous by means of the reconstruction error of their projection using the major components. Similarly, uncorrelated records are detected using the minor components. Other approaches that use PCA to exploit the correlation structure of the data are described in [175] and [141]. Both apply PCA to entropy features, considering the minor components as indicators of how well the normality is represented. In [175], the residual projection of the data obtained by means of the minor components is used to identify anomalies with a high reconstruction error. In all these cases, the number k of major components is selected at training time. In contrast, in [141], the authors use the angle between the PCA projections of two contiguous non-overlaping windows of data to obtain an anomaly score and, to do so, the number of major components k is tuned at run time, selecting the number that minimizes the orthogonality between the normality of both windows.

Time series analysis has also been used for unsupervised anomaly-based NIDS. These statistical methods are used to predict future values for different network features (SFD or MFD), which are compared with actual, measured values. Records with high prediction errors are flagged as anomalies [52, 305]. Time series have

also been proposed to model the likelihood of the alarms generated by an unsupervised Long Short Term Memory Neural Network used as anomaly detector [5]. The method filters the number of alarms, indicating that there is an anomaly when this number deviates highly from the expected value.

We end this description of unsupervised anomaly-based systems with clustering methods. They group data that share similar characteristics with the help of similarity or distance measures [301]. The objective is to identify large or dense clusters, which usually correspond with normal data, flagging as suspicious anything that deviates from those clusters. Those methods assume that there is a minority of attack samples, compared to normal traffic. If this were not the case, expert knowledge would be required to interpret the obtained groups. Depending on the clustering method, thresholds can be applied to density measures [236, 94, 16, 59, 82, 45, 220], flagging observations in low density zones as attacks, or to distances [170, 250, 220], flagging farthest observations (or clusters) from the reference normal points as anomalous.

Supervised approaches have also been proposed for anomaly detection in NIDS, although rarely. In [326], a Random Forest classifier is trained with normal data, using the service identifier (TCP, UDP port number) as the class label. In operation, when a new sample arrives, if the service assigned it by the classifier does not match the specific service to which the sample belongs, an alarm is raised.

Weakly supervised anomaly detectors for NIDS use partially labeled data, aiming to improve the accuracy of unsupervised methods. Proposals [86, 220] apply a one-class SVM classifier using a dataset in which only some samples are labeled as "normal traffic". In [106], a multi-classifier system formed by various ensembles of one-class SVMs is used to model normal particularities of network services. To do so, each ensemble, formed by two to three models using different feature subsets, is trained for each service. Similarly, proposal [233] also uses an ensemble of one-class SVMs, although it is focused on HTTP attacks. As in the previous approach, each model is trained with a different set of features, and the output of all the models is combined to obtain the final classification result. In [19], an ensemble of five Hidden Markov Models (HMM), a probabilistic classifier, is used to predict the probability that a new traffic record is normal. To do that, only payload sequences of normal HTTP traffic are used to train the models. Again, their output is combined to obtain a final score. In [114], a one-class SVDD (Support Vector Data Description) algorithm that uses active learning strategies is proposed. In that approach, a small portion of very uncertain data is labeled (by an expert) in order to train the detector and increase the detection performance. Proposal [181] uses a semi-supervised variant of SOM [168] in which the class information of the labeled records is used to assign labels to clusters. At training time, when a new sample is fed, it is labeled with the class of its nearest cluster (known as the activated cell). Both, that and the remaining clusters that have the same class label, are then "rewarded" moving their centroids closer to the sample. Semi-supervision is added to deal with unknown clusters (those that are unlabeled), which are always rewarded. The method is also able to merge and split clusters, respectively merging very pure cells (those

with a large number of records of the same class) or creating new clusters from impure cells.

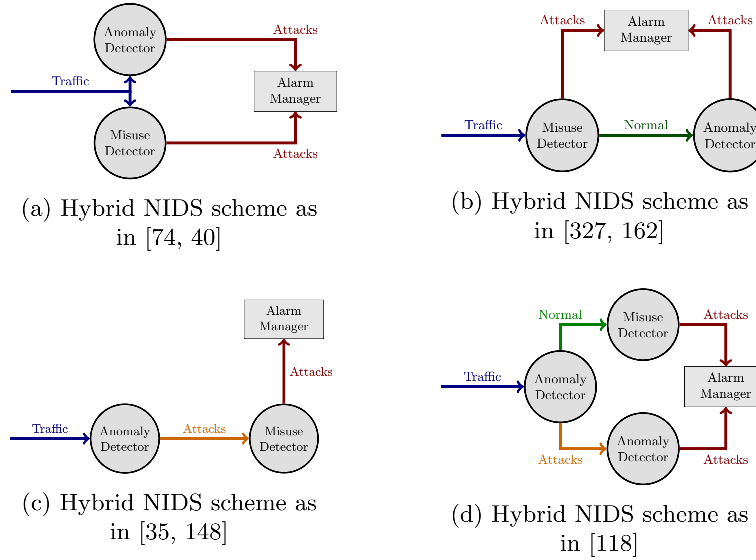
The anomaly-based NIDS methods discussed above present some advantages and limitations which, although they are not enough to perform a comparison among them, they are still worth mentioning. For clustering, the use of feature transformation methods depends on the choice of the distance and the chosen clustering algorithm. Density methods create groups of any shape, while distance based methods form spherical clusters [2]. Correlation methods require numerical features for the computation of covariance or correlation matrices but, as their main shortcoming, they lack explainability when attacks are detected, as those matrices are generated from groups of records. Time series analysis demands using numerical data and existing methods for NIDS are difficult to apply with non-stationary data. However, it is a useful resource to model temporal relations in traffic [2]. Some frequency-based methods are also able to model temporal relations while working with any kind of data, but they may have poor performance in networks with unbalanced types of normality. Finally, weakly supervised methods also require numerical features to work and are not as easily adaptable to incremental approaches as the other methods because they would require constant re-training steps. Instead, they have better accuracy than unsupervised methods because are assisted by labeled data. The cost of obtaining labels is their main shortcoming, but this effect can be reduced by means of active learning techniques [114].

2.7.1.3 Hybrid NIDS

Misuse and anomaly-based methods can be combined into a hybrid detector trying to overcome the limitations of each method separately. At the cost of an increased complexity of the NIDS, this combination can be done in different ways, as depicted in Figure 2.5.

In Figure 2.5a we can see a simple case in which a misuse-based detector and an anomaly-based detector work in parallel, flagging an observation as malicious if it causes an alarm on any of the two components [74]. The detector described in [40] incorporates both misuse and anomaly approaches in a single data mining method. The system outputs the classification made by the misuse component, and computes the level of deviation from the identified class. Note how this parallel set-up may detect more attacks than a single system, at the cost of an increase in the number of false positives caused by any of the two detectors.

Another way of combining two detectors is concatenating them. The first module of the detector receives traffic records and feeds its output to the second module. Depending on the order in which the misuse and anomaly based detectors are chained, the combination reduces the number of false alarms or increases the detection ability. The architecture proposed in [327, 162] places first the misuse-based system, which receives all network records and identifies well-known malicious behaviors (see Figure 2.5b). Afterwards, the anomaly detector receives only those records previously classified as normal in order to discover unusual patterns, increasing the detection ability of the NIDS. The inverse approach has been proposed



(a) Hybrid NIDS scheme as in [74, 40]

(b) Hybrid NIDS scheme as in [327, 162]

(c) Hybrid NIDS scheme as in [35, 148]

(d) Hybrid NIDS scheme as in [118]

Fig. 2.5: Hybrid-based NIDS schemes.

too [35, 148], as depicted in Figure 2.5c, with the intention of reducing false alarms. The anomaly detector flags suspicious traffic, and afterwards the misuse module confirms and explains the flagged anomalies.

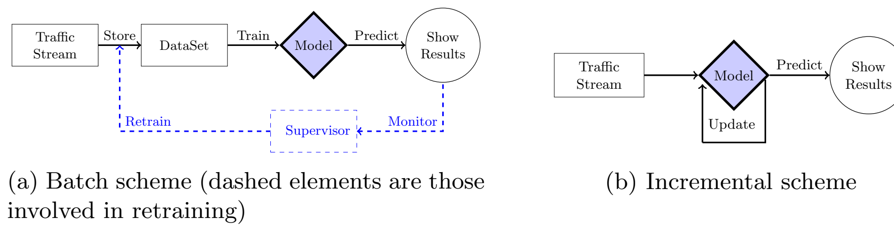
The variant depicted in Figure 2.5d, and proposed in [118] puts an anomaly detector first. Samples flagged as normal are forwarded to a misuse detector that may identify attacks that passed undetected. Suspicious samples are forwarded to a second, slightly different, anomaly detector, aiming to confirm the attacks and to reduce the number of false alarms. Note that this is the most costly scheme as all input records are analyzed twice.

2.7.2 Batch vs. Incremental Learning

In our taxonomy, the second NIDS classification criterion refers to the different policies used to learn models and keep them updated. Networks, and network traffic, evolve continuously and, therefore, the characteristics of normal traffic and attacks change over time [271]. In this context, the flexibility and adaptability of the data mining algorithms and their tolerance to previously unseen patterns is crucial [179]. However, this fact is rarely mentioned in the literature. In this section we study the performance and practical usability of two different learning approaches discussed in the NIDS literature: batch and incremental.

2.7.2.1 Batch learning NIDS

Learning in *batch mode* means that the data mining algorithm is applied once to a set of static (training) data, labeled or unlabeled, and after that, the results



(a) Batch scheme (dashed elements are those involved in retraining)

(b) Incremental scheme

Fig. 2.6: Learning schemes of data mining methods.

or detectors obtained are used to make predictions for new incoming data (see Figure 6.1). Typically, such algorithms require multiple passes over the training data and, since they have not been designed specifically for streaming scenarios, they are usually costly in terms of both time and memory requirements [157]. The effectiveness of these models is high at deployment time, but they do not evolve with time. There is, thus, an implicit assumption of data being generated by a stationary distribution [270]. This is an important drawback, as it seriously challenges the ability to detect previously unseen traffic patterns, that may correspond to new or mutated attacks, or to new kinds of normal traffic [292].

Most classical data mining methods, independently of their degree of supervision, are based on batch learning. Many NIDS proposals that leverage off-the-shelf supervised classification or clustering algorithms belong to this category. For example, [14] and [257] rely on well-trained models to perform misuse detection. In a similar fashion, [283] and [220] use a batch approach to model normality in their anomaly detectors. None of these proposals are able to update their detectors automatically.

Other proposals try, instead, to somehow capture and adapt to the traffic changes via retraining [102]. This extension of the batch mechanism adds a supervisor component to the detection framework (see dashed elements of the Figure 6.1) that monitors the status of the network or the response of the system, and decides whether it is necessary to trigger a retraining procedure. Retraining means using the data mining algorithm of choice again, but with a recent batch of stored records, to generate an updated model that replaces the obsolete one [102]. All detectors built using batch learning can be enhanced with retraining. However, this learning mode also has some drawbacks. The selection and storage of the data required to build an updated detector in each retraining phase and, when supervised methods are in use, the labeling of this data (see Section 2.4.2), are non-trivial tasks that require additional resources [292]. Also, as network traffic changes at non-predictable times, additional complexity in terms of a monitoring mechanism must be put into the NIDS.

Some examples using retraining have been proposed in NIDS. In [59], basic time series analysis is carried out over statistics of grouped flows in time windows (MFD features) to detect changes in flow patterns. Then, for any time window in which changes are detected, a retraining is done over the flows within the time window,

using their SFD features. A density-clustering algorithm is applied to rank records by their distance to the largest cluster. Finally, the records farther than a predefined threshold are flagged as anomalies. In [227] a supervisor is added to the proposed NIDS that monitors network alterations such as changes in open ports (available services) or connected devices. It is assumed that those alterations modify the structure of normal traffic, but attacks do not vary. When a change is detected, an auto-encoder is used to learn a new sparse representation of the data reflecting the new traffic characteristics. The same labeled dataset used for the initial training is transformed with the new auto-encoder to build an updated classifier that replaces the previous one.

2.7.2.2 Incremental learning NIDS

Incremental learning algorithms assume that (1) data is unbounded and arrives in a continuous manner, and (2) data may be generated by non-stationary processes and, therefore, may evolve over time [270, 219]. Consequently, learning is done continuously with the arrival of new data samples (see Figure 2.6b). The main difference with retraining strategies is that the model or information stored by the algorithm is updated continuously, instead of being rebuilt from scratch. Adaptation to changes is gradual, which better suits highly variable scenarios such as networks, where stationarity is infrequent [287, 102]. In general, the cost is also lower than that for retraining as less data is incorporated to the model. On the downside, since they learn and provide a response in an online fashion, these algorithms can usually only perform a few passes over the data and they have time and memory limitations [102]. Another disadvantage of incremental learning is convergence time: learning about changes requires some time and, meanwhile, accuracy may be poor. This is particularly true at the initial stages of deployment or when abrupt changes occur [101]. Continuous adaptation also makes these systems sensitive to noise and perturbations such as those intentionally generated to evade detectors [165]. As such, incremental approaches require additional and specific evaluation measures that monitor the evolution of the learning process and guarantee its correct behavior before deployment (see Section 2.8.1).

As the most common approach, we can find a group of unsupervised methods that analyze if each incoming data point in the stream is anomalous, that is, if it is dissimilar to other data points observed previously in the stream. Since they are incremental algorithms, these methods typically only process each data point once, summarize the stream in a representation with small memory requirements that can be easily updated in an online manner, and compare the incoming points with this representation. A very naive approach within this group is described in [223]. The attribute-value pairs of network records are directly analyzed, flagging data points with infrequent values in certain attributes as anomalies. The stream is thus represented as a table of frequencies of different values for SFD, MFD and payload features, which is updated incrementally with each new sample.

As a slightly more complex approach that follows the same idea, several NIDS papers describe the use of incremental clustering algorithms. These algorithms

group the data with similar characteristics and identify attacks as observations that lie far from the existing clusters, or that are located in sparse clusters. Contrary to batch clustering approaches, which group static sets of data into a fixed set of clusters, these methods are designed for streaming environments. As such, the data and cluster structures are represented in different compact manners, i.e., by storing only the objects of the current time window [16], by saving only prototypes or centroids [86, 250], by arranging data into groups of similar data objects (microclusters) represented by a vector of characteristics (mean, variance, etc.) [94, 45], or by using a grid structure in which the data is arranged and summarized using some statistics [82]. Cluster assignments are made online and, additionally, in order to adapt to changes that can happen in the distribution of the data, some algorithms include mechanisms to update cluster assignments by adding and removing clusters as observations arrive. For example, proposal [94] uses a nature-inspired (bird flocking) optimization algorithm to control the evolution of the clusters over time, allowing merge or split operations of microclusters. In [45] microclusters are represented as Gaussian Mixture Models (GMMs) that can be compared using the Kullback–Leibler divergence and merged if they are sufficiently similar. The approach followed in [250] uses the distances of a sample to its two nearest prototypes to measure the uncertainty of the NIDS outcome in an active learning setup. When distances are very similar, the human manager is required to take corrective measures over the system (select the correct prototype, request a new one, merge both prototypes, etc).

A second less common group of unsupervised incremental NIDS uses the temporal information of the stream to identify attacks. Their objective is to incrementally learn and update a model that will somehow describe the normal evolution of the stream. Any observation that does not follow the expected behavior will be flagged as an attack. The difference between the proposals in this group lies mainly in the method used to model the evolution of the stream. Proposal [120] leverages the incremental clustering algorithm introduced in [86], in which a new cluster is created each time that a sample lies outside the limited area (using a fixed radius) of current clusters, and analyzes the transitions between cluster assignments of consecutive traffic samples. Another example is [5], where a Long Short Term Memory (LSTM) network is used to model the evolution of the traffic stream. The LSTM accounts for the transition frequencies provoked by consecutive network samples in the cells of its architecture, updating the transitions with each new sample. In both cases, uncommon transitions are flagged as possible attacks.

Supervised incremental methods have also been proposed in NIDS, although rarely. An example of incremental supervised methods is [140], where an online version of the Adaboost algorithm, a classifier based on an ensemble of weak classifier models, is used. The method updates those weak classifiers with poorest performance each time a new data sample arrives. Labels for those samples are derived by a NIDS classifier that is retrained at fixed intervals. A NIDS model based on Reinforcement Learning is proposed in [192]. The method uses an incremental Neural Network that is updated (rewarded) with every prediction. Other examples include [292, 293], which use an ensemble of Very Fast Decision Trees (VFTD), an algorithm

that learns classification trees incrementally using small batches of data under the assumption that classes do not evolve with time [142]. Again, the main drawback of this kind of methods is the requirement of class labels at operation time. Note that proposals [140, 192] assume that the actual label of all the samples is known during the update phase (the reinforcement phase in RL frameworks), something that may be problematic (cost, latencies, mistakes during the labeling, etc.). When labels are not needed for all the new incoming samples, a possible solution is to select the best candidates for labeling (as explained in Section 2.4.2) by means of active learning techniques [151]. Such techniques have been applied in [292].

In general, incremental approaches (supervised or unsupervised) lack maturity in the early stages of learning (when models have not yet been fed with enough data) [101]. To overcome this problem, some off-line data may be provided before deployment. An initial model is trained using batch learning, thus assuming an initial distribution of the data. After that, the model is incrementally updated as new samples arrive [141, 293, 250, 45, 292]. This approach has the advantage of relying on a well-learned detector in the initial phases, while overcoming the limitations of pure batch processing.

	Misuse	**Hybrid**	**Anomaly**
Batch	[225, 159, 218, 149, 294, 137, 27, 13, 123, 15, 14, 161, 320, 163, 257, 288, 72, 266, 264, 107, 213, 259, 166, 49, 185, 230, 186, 262]	[162, 327, 40, 148, 35, 74, 118]	[198, 236, 283, 176, 319, 223, 267, 326, 52, 305, 19, 181, 170, 233, 167, 106, 86, 173, 175, 220, 114]
Re-train	[227]		[59]
Incremental	[293, 140, 292, 192]		[16, 120, 94, 82, 45, 60, 141, 250, 312, 5]

Table 2.5: Classification of surveyed NIDS proposals, arranged by learning and detection approach.

Most incremental NIDS methods are based on unsupervised approaches (see Table 2.5). The main reason is that labels are not always easy to obtain, especially in real time. Unsupervised methods used in NIDS base their detection mechanism on thresholds (over frequencies, densities or distances). Those thresholds are generally fixed, even though these methods assume that traffic data evolves over time. As such, this evolution may not be captured adequately and this may result in a rise in the false positive or negative ratios [16, 120, 250]. In some cases, it is assumed that the thresholds follow some statistical distribution, used to dynamically estimate the confidence interval in which most normal observations should be found [5, 45]. Care must be taken with some distributions, such as the Normal, that are sensitive to outliers, as attackers may use this characteristic to evade detection, e.g., generating a high number of anomalies to widen the threshold and make attacks undetectable.

A small number of NIDS proposals are based on supervised incremental methods (see Table 2.5). They should be able to learn to differentiate normal from malicious activities with the arrival of new (labeled) data. However, some supervised algorithms have difficulties adapting to changes in the distribution of the data [292]: they can adapt to the appearance of new classes in the data, but not to modifications of the characteristics of previously known classes. The adaptation rate depends on the specific characteristics of the algorithm. It can be slow unless a change-detection component is added, which would help to quickly forget the characteristics of old data [102].

2.8 Evaluation Stage

The objective of the evaluation stage is to measure how well the previous stages have performed. Ideally, this evaluation should be as rigorous as possible, using different criteria such as detection accuracy, complexity, adaptability, understandability, security, etc.

It is not easy to define valid metrics and comparison criteria for all these characteristics. Let us focus on complexity, very closely related to the ability to deploy a NIDS in a real environment. Only a few papers report complexity computations, and we find them incomplete [19, 162, 123, 250]: computational complexity of the data mining algorithm in use defines only a part of the cost of the system. The remaining steps of the KDD process also have a cost, and implementation issues (such as programming languages used, compilers, libraries, run-time environments etc.) play an important role in the processing ability of the NIDS. A fair evaluation of all these aspects is an extremely complex exercise that requires a comprehensive testing environment –something that is beyond the scope of this survey. Similar limitations could be stated for adaptability, understandability and security.

The predictive performance of a NIDS, however, can be measured, with many different metrics that have been proposed. These metrics are quantitative measures that are commonly used as yardsticks to compare NIDS proposals. This is why most authors have mainly paid attention to this performance criterion, and that is why we do the same in this review. Things are easier when comparing proposals following the same learning paradigm (either batch or incremental). However, note that different aspects have to be considered when evaluating batch and incremental models. Thus, making their comparison more difficult.

2.8.1 Validation Frameworks

In a typical batch setting, where a static distribution of the data is assumed, k-fold cross-validation is a common approach to estimate the true predictive error of a model. Cross-validation (CV) is an honest performance estimation framework, where the dataset is randomly split into various subsets or folds, and the error estimate is calculated as the average of the results of the evaluations performed over the

different folds [128]. Cross-validation entails several incompatibilities with the online and incremental framework: (1) samples are assumed to be time-dependent and (2) incremental models evolve as new data arrives [43]. As folds in cross-validation are selected at random and samples are assumed to be independent, the progressive evolution and time-dependencies of samples are broken, i.e., the evaluation does not reflect the detector's ability to deal with concept drift [101].

In incremental CV frameworks, the data folds used for training and evaluation are created in an online manner. There are a few variations but, in the most simple approach, each time a sample arrives, it is randomly assigned to one fold. A model is learnt with each fold and incrementally updated every time its fold receives a new sample. As in standard CV, each model is evaluated using all the folds not used to train it. The final performance metric is obtained as an average of the metrics calculated for all the models. A slightly more complex approach assigns the new incoming sample to $k-1$ folds and updates the corresponding $k-1$ models. All models are evaluated on the data that has not been used to learn them (assigned to their fold).

A different approach is called *prequential* evaluation. Every time a new sample arrives, the prediction made by the model is compared against the actual label of the sample. After this evaluation, the model is updated using the sample. The final prequential error is computed as the accumulated sum of all the errors obtained. Incremental cross-validation and prequential evaluation can be combined as k-fold prequential evaluation. The result is a more robust error estimate [43].

All these approaches assume that the labels of samples are known beforehand. They can deal with partially labeled data, computing the error using only the labeled samples [101]. However, note that when very few records contain labels, the estimation of the error might be poor, as the approximation of the error converges to the true error as more labeled data is available [101, 43].

A characteristic problem of the previous incremental error estimation methods is the high number of errors reported at the initial stages of learning, when the models are immature. Also, the number of errors may increase due to concept drift. To reduce this effect, forgetting mechanisms, such as sliding windows and fading factors, have been proposed. With sliding windows, an error estimate is computed for the most recent k samples, while fading factors apply a weight to discount older information from the computation of the error estimate [101]. In both cases, the chosen window size and the decay factor parameters are critical to perform fast detection, i.e., rapid adaptation of the model to recent changes [100, 101].

2.8.2 Evaluation Metrics

In terms of ML, predictive performance metrics measure how well the predictions provided by an algorithm match with the true labels of the input samples. The confusion matrix, see Table 2.6, is used to easily visualize the detection performance of a NIDS. For binary detection problems (normal vs. attack), it is common to see the positive class as attacks, while negative means normal activity. For multiclass problems, where more than two classes of traffic can be detected, it is common to

		Actual class	
		Positive	**Negative**
Predicted	**Positive**	True Positive (TP)	False Positive (FP)
	Negative	False Negative (FN)	True Negative (TN)

Table 2.6: Confusion matrix. Rows correspond to the predicted class and columns to the actual class. The value represented in each position of the matrix indicates the number of instances predicted, as stated by the row, that actually belong to the class indicated by the column.

Metric	Formula	Metric	Formula
DR, TPR, Recall, Sensitivity	$\frac{TP}{TP+FN}$	Precision	$\frac{TP}{TP+FP}$
FAR, FPR, Specificity	$\frac{FP}{FP+TN}$	Accuracy	$\frac{TP+TN}{TP+TN+FP+FN}$
TNR	$\frac{TN}{FP+TN} = 1 - FPR$	Error Rate	$\frac{FP+FN}{TP+TN+FP+FN} = 1 - Acc$
FNR, Miss Rate	$\frac{FN}{TP+FN} = 1 - TPR$	F-score/F-measure	$2 * \frac{Precision * Recall}{Precision + Recall}$

Table 2.7: Common evaluation metrics derived from the confusion matrix.

use a *one-vs-all* procedure: the positive class is the one that we want to measure, while the negative class groups all the instances of the remaining classes.

As can be seen in Table 4.3, several metrics (that receive different names in different works) can be derived from this matrix, each one focusing on a different facet of the prediction capacity of a NIDS. Note that some of them, such as the Detection Rate (DR), the True Negative Rate (TNR), Accuracy and Precision, must be interpreted as *the higher, the better*, taking values close to 1 when the detector performs very well. Most of these metrics account for the proportion of samples for which the model predicts its class correctly. In contrast, the best detectors should report values close to 0 for the False Positive Rate (FPR) and the False Negative Rate (FNR), as they represent failure ratios.

Another common metric is the ROC curve. It visually shows the trade-off between the TPR and the FPR of classifiers, by means of a bi-dimensional plot, for different values of the decision threshold. The area under the ROC curve (AUC) is also used as a metric to evaluate NIDS.

Not all the metrics used by the machine learning community are meaningful for the specific NIDS problem. Network traffic is a highly unbalanced scenario in which normal traffic is much more common than attacks. Moreover, network attacks may have a severe impact on the victim and, thus, metrics used for NIDS should well reflect the effects of not detecting an attack (false negatives). A robust metric in unbalanced scenarios is the Kappa statistic. It quantifies the behavior of a predictive model in contrast to a random chance detector [66]. The kappa statistic can be seen as a correlation coefficient between the predictions of the model and the true labels,

taking a value of 1 when the predictive model always guesses the class of the samples correctly, 0 when it behaves as a random chance detector and -1 when it fails all the predictions.

The most common metrics reported in the surveyed NIDS literature are the DR (Detection Rate) and the FPR (False Positive Rate), as they inform about complementary detection abilities. The DR measures how well the system identifies abnormal behaviors as attacks, while the FPR measures the ratio of normal observations incorrectly flagged as attacks. The goal of a good IDS is to maximize the former while minimizing the latter. Summary metrics that combine others (such as the F-measure) are also used, but should be considered as supplementary information due to its poor explainability. ROC curves are useful as they provide a graphical way to compare detectors [82, 45]. However, when plots are very similar, additional metrics may be needed for further clarification. Although being a useful indicator for NIDS, where imbalance is common, the Kappa statistic is not reported in the literature. This fact contrasts with the use of some metrics, such as the Accuracy, which do not properly reflect the performance in unbalanced scenarios [258], and has been reported in several works [35, 213, 185, 148, 320, 230].

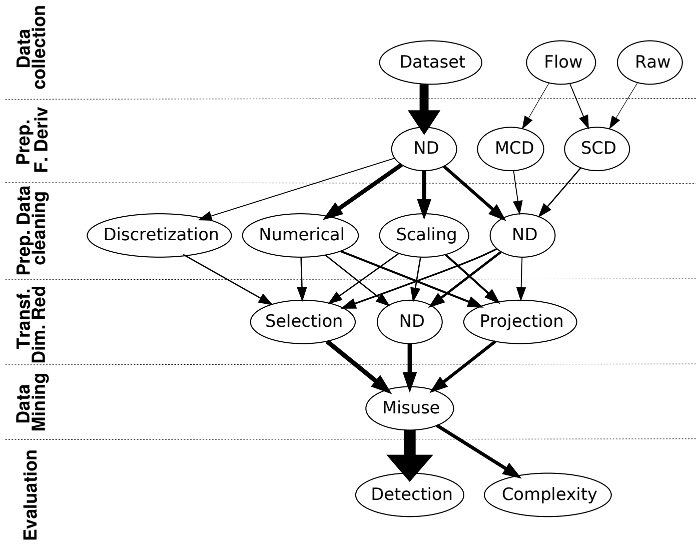
2.9 Discussion and Conclusions

An ideal NIDS should fulfill a large set of desirable characteristics including, among others, the following: early detection of attacks, high attack-detection coverage (ability to detect the maximum number of attacks), high detection rate with low false alarm rate, ability to scale according to network requirements, and robustness to attacks targeting the detector [105, 53]. In this section we summarize the conclusions extracted from our literature review.

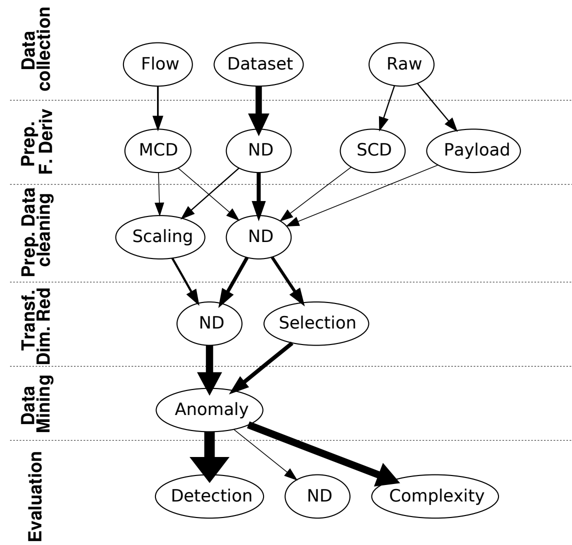
In order to have a general view of the use by NIDS authors of the techniques discussed in this chapter, we have rendered two graphs (see Figures 2.7a and 2.7b) that represent the paths followed when designing (and evaluating) a NIDS, for both misuse and anomaly-based systems. In these graphs, each level corresponds to a different phase of the KDD process, and the nodes within each phase correspond to the methods used in the surveyed articles. Nodes are joined by arrows, whose thickness represents the proportion of works that apply a particular (destination) method after the (source) method of the previous phase. To allow a better understanding of the graphs, we have simplified them by removing nodes for methods with low utilization (below 5%). In general, note that main paths (sequences of thick arrows) are characterized by the presence of acronym “ND” (method Not Discussed for that phase), due to two main reasons: (1) many articles do not mention the methods applied to the data, and (2) oftentimes, data mining algorithms are applied directly to “cooked” records obtained from public datasets.

2.9.1 Reproducible Research

A first conclusion of our literature review that can be inferred from the previous paragraph is that most papers provide insufficient detail about their pipeline and,



(a) misuse approaches



(b) anomaly approaches

Fig. 2.7: Paths along KDD of misuse and anomaly surveyed approaches. Levels represent the different phases of KDD. Nodes represent methods used at each stage. Edge thickness depicts the proportion of the surveyed works using the method below.

more specifically, about the procedures used for feature derivation, transformation and validation. In some cases, this is due to the use of public datasets in which all this work has already been done. However, we have found several articles that apply data mining algorithms that can only deal with numerical data over the set of heterogeneous records provided by a public dataset, without specifying any feature removal or feature transformation method [259, 181, 186, 5, 45, 250, 227]. In some other cases, transformations are vaguely described: “we transformed categorical variables to a numerical representation”, without stating how [40, 266]. Such a lack of details about the procedures carried out to build NIDS hinder their analysis, evaluation and comparison. A fact that has been reflected in our analysis, where some discussions may seem superficial given the difficult to properly find the grounds behind the use of some techniques for NIDS.

2.9.2 Data Collection, Feature Extraction and Public Datasets

Graphs 2.7a and 2.7b show how, in the first phase of KDD (data collection), most NIDS use public datasets for both model construction and evaluation purposes. In Section 4.4, we have already explained how these datasets were generated, also pointing out their deficiencies (short capture periods, discontinued captures, uncertainty during the labeling process and lack of realism, among others). A NIDS that works successfully with any of these datasets is not guaranteed to operate equally well in real scenarios. Thus, it is not possible to perform a proper evaluation of NIDS with a single dataset; the use of various datasets, with different characteristics, has to be considered. Ideally, datasets which satisfy most of the mentioned characteristics, and that reflect the current use of the network by applications/nodes, are desirable. These databases should be captured at different network locations, and should thus provide different views of the network.

Another potential weakness related to the use of public datasets lies in that each one provides a particular set of already derived features, and most researchers work only with them. Available features may be useless when trying to detect newer attacks, or may contain data that is impossible to obtain in real scenarios, for example features annotated by human experts. To guarantee the derivation of custom features and provide more flexibility to researchers, the availability of raw packet data in public datasets is a must. Indeed, a few NIDS authors derive their own set of features from raw data (see Figures 2.7a and 2.7b), normally SFD, MFD or payload features, that have shown to improve the accuracy of the detector [136].

Regarding payload features, we have already mentioned that the trend is that most applications interchange data in a secure, encrypted form. Nonetheless, most public datasets do not contain this kind of traffic (another reason that makes them unrealistic). This absence of encryption has allowed the extraction of payload features, useful to detect some attacks that leave their footprint in the content of a connection. However, either ciphered or in cleartext, obtaining these features is barely possible in real world high-speed networking scenarios.

Precisely because dealing with payload data is anything but easy, many authors have limited their detectors to use SFD and MFD features in order to identify point

and collective malicious patterns. However, these derived features are usually very simple: flow, packet or byte counts and average rates.

2.9.3 Feature Transformation and Dimensionality Reduction

Our graphs (see Figures 2.7a and 2.7b again) show how feature scaling methods or transformations of categorical features into numerical representations are more common in misuse systems than in anomaly detectors. The main reason is that some data mining algorithms (mainly supervised classifiers) require datasets which are free of outliers, balanced in terms of labels and homogeneous in terms of data type natures (numerical or categorical) to work properly.

Numerical transformations of categorical data allow the use of data mining algorithms that can only deal with numerical data. Some of these representations, such as dummy features or frequency encoding, substantially increase the size of the feature vectors. Consequently, dimensionality reduction methods, such as FSS and projections, become mandatory. Some anomaly-based NIDS performs this reduction through the manual removal of unwanted features.

This combination of techniques is used to enable the use of data mining algorithms and also to increase their detection accuracy. While this may be a good solution for static environments, what happens when network traffic changes? New threats may leave traces in features incorrectly modified (discretized or scaled, with unknown categories...) or selected for removal, as this was done using criteria based on obsolete traffic patterns.

2.9.4 Data Mining Algorithms and Learning Schemes

As regards the data mining phase, misuse systems are mainly based on supervised classifiers, while unsupervised and weakly supervised methods are the choice for anomaly detection. Based on our review of the literature, we have seen that many works based on supervised classifiers have erroneously used the term “anomaly detection” to refer to their detection approaches. In view of this, we have concluded that the existing definitions for the concepts of misuse-based and anomaly-based NIDS do not allow a clear categorization of the existing NIDS, and in this work we have provided a new taxonomy to avoid such miscategorizations.

Also, we have pointed out the limitations of batch learning methods when dealing with evolving traffic patterns. Despite this, we have shown that most published works use batch learning, implicitly assuming static and unrealistic traffic patterns. Sometimes retraining mechanisms are implemented and the training of the new model is done while the NIDS is operating with the obsolete model, which in fact is dangerous as the number of false alarms may increase, while attacks may pass undetected [53]. The number of incremental learning methods that consider networks as continuously evolving scenarios is very small for both misuse and anomaly detectors.

The small interest paid by the authors to temporal dependencies in the input data when designing detectors also needs to be highlighted. A few works apply elementary time series analysis to exploit temporal relations, but a thorough study of the temporal relations is missing. Most proposals analyze traffic data once connection statistics have been computed, i.e., detection is carried out over connections that have already occurred; therefore, an attack is detected after it starts taking place.

2.9.5 NIDS Evaluation Beyond Accuracy

A common issue when comparing NIDS papers is the lack of an adequate framework to evaluate and compare different proposals. Authors often use unfit metrics (we have already discussed how some metrics that are commonplace in ML literature are meaningless in NIDS) and obviate the evaluation of important properties beyond those related to accuracy. Complexity, adaptation ability and resistance to attacks towards the NIDS itself should be assessed.

Complexity is sometimes discussed in the NIDS literature (see the lower part of Figures 2.7a and 2.7b) but, in our opinion, this issue deserves further research efforts. A theoretically excellent detector is not practical if response times are high and attacks are reported only after the attacker has fulfilled his/her purpose. An effort to collect the *training* complexity of NIDS proposals was made in [53], but the computational cost associated to this phase is only critical for retraining procedures. In addition, and as we have seen throughout this survey, the operational cost of NIDS detectors may be dependent of any other steps apart from detection (data preprocessing, data transformation...). Such costs should be reported, as they would indicate whether or not a NIDS may work in real time, and with how many resources.

The evaluation of the adaptability of incremental learning approaches is also ignored [172]. These learning mechanisms need to be robust to noise (or attacks) while being able to adapt to newer traffic patterns. In other words, a balance between sensitivity and robustness has to be found by means of a proper evaluation of different variability scenarios, a fact barely discussed in NIDS that use incremental learning.

Finally, assessments of the security features of NIDS proposals are also infrequent. As can be seen in Figures 2.7a and 2.7b, there is no node representing this aspect of the evaluation. A NIDS plays an important role in the protection of the corporate network, and should be resistant against attacks that target it [28]. To the best of our knowledge, the first article discussing this risk dates from 2006 [37]; posterior research also tackles this issue [217, 46, 228, 136]. After that, a reduced number of NIDS proposals incorporate some protection mechanism, but of limited scope or without performing any evaluation [19, 305, 293]. More effort must be placed on researching NIDS security, as well as on applying a framework to evaluate it [47].

Part II

On the use of AI algorithms for Android malware detection based on static analysis

3

Background

The second part of this thesis focuses on Android malware detection using AI techniques, and more specifically, machine learning (ML) algorithms. To perform the detection of Android malware using ML, Android apps need to be preprocessed to extract the set of features that best describe their behavior. This chapter presents basic information required to understand the contents of the following chapters. We first describe what the structure of an Android app file is and the type of information that can be extracted by statically analyzing its contents. Then, we detail how supervised classifiers, a type of ML algorithms, are used to perform the detection of Android malware using static analysis features. Finally, we describe what obfuscation techniques are, and how they can be used to hinder static analysis and avoid the extraction of the features used by detectors.

3.1 Android Apps

An Android app is distributed and installed via an Android Application Package (APK) file, which is a compressed (ZIP) file containing all the resources needed (e.g., code, images) to execute the app. Figure 3.1 shows the internal structure of an APK file. During the building process, the APK file is signed with the private key of the developer. To validate this signature, the APK includes the public certificate of the developer inside the *META-INF* folder. This mechanism guarantees the integrity of the APK[1]. In a nutshell, before installing an app, Android verifies if the files in the APK match a pre-computed signature and continues with the installation only if the integrity check succeeds.

Android apps are generally developed in Java or Kotlin (Google's recommended language for Android development) and transformed into Dalvik bytecode format during the compilation process. The Dalvik bytecode runs on the Android virtual

[1] Note that Android does not verifies the validity of the developer's certificate but instead, uses this mechanism to validate the integrity of the content within the APK. Therefore, the developers' certificates can be self-signed.

machine, which serves as a platform-independent environment. Interactions between the hardware components of mobile devices, directly managed by the operating system, and the apps, managed by the virtual machine, are performed through API libraries. These APIs provide a common way to access to the hardware capabilities required by the apps. Thus, abstracting the programmer from the particularities of devices.

The Dalvik bytecode of apps is located inside the APK in the classes.dex file. An Android app can contain one or multiple DEX file(s) (i.e., *classes*.dex*). Each *.dex* file can reference up to 64k methods [76], such as the Android framework methods, other library methods, and the app-specific methods. It also includes constant and variable definitions.

When an app has to meet very strict performance constraints, or interact directly with hardware components, Android allows developers to introduce native components written in C and C++ (i.e., *native code*). For the native components, Android provides an Android Native Development Kit (NDK) [248] that generates native libraries in the form of Linux shared objects. Such objects are stored into the *lib* folder.

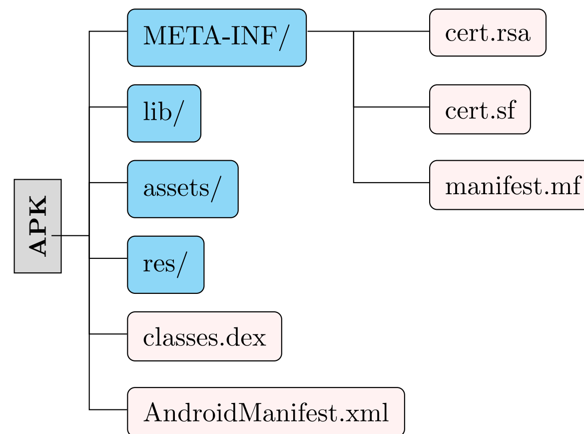


Fig. 3.1: Structure of an APK file

The *AndroidManifest.xml* file of an Android app defines a set of properties and components that the app requires from the platform in order to work. This file is in XML format and it is divided into three main blocks or sections:

- The *Application Components* block defines the elements of the app that interact with the OS during the app execution or when a specific action is requested to the OS by the user. These components implement background functionality (services), manage user screens and app interactions (activities), enable app

interactions with other OS components or apps (broadcast receivers and intent filters) and the interfaces to share data with other processes (content providers).

- The *Hardware and Software Features* block defines the OS properties and functionalities that the app requires to function. This includes software features such as backup support, user account management, input methods, etc. Hardware features include elements such as the camera, bluetooth transmission, fingerprint sensor, etc. Declaring requested features is useful, for example, to prevent an app from running on a phone that does not fulfill the required specifications.
- The *Permissions* block indicates the features that are required by an app but are protected by the OS. Access to these functionalities must be explicitly granted by the user. By default, Android apps do not have permissions to perform actions that could compromise the OS, user information, or other apps. For example, to access to the microphone, camera, contact list, Internet connection, location, etc.

Finally, the *res* folder contains the compiled resources (e.g., images, and strings), and the *assets* directory includes the raw resources, providing a way to add arbitrary files such as text, HTML, font, and video content into the app.

3.1.1 Static Analysis of Android Apps

As will be explained in the next sections, in order to use ML methods for malware detection in Android, the apps have to be described/characterized by means of a set of features. In this sense, static analysis is a software technique that inspects apps to study their characteristics without needing to execute their code and monitor their behavior at runtime [183]. Ideally, by means of static analysis, all execution paths present in the code and all the information in the files of an app can be inspected. This is done by using tools with code inspection and interpretation mechanisms that extract understandable structures and data describing the internal functions of apps. These include tools such as AXMLprinter[2] that enables access to fields and components declared inside the *AndroidManifest.xml* file, and *bakSmali*[3], that decompiles and converts the classes.dex file(s) to a higher level (more readable) format.

Through static code analysis, different features can be obtained, including information about instructions, methods, classes, strings and the usage of API calls [88]. It is also possible to build different graph structures representing the code. These include Call Graphs, built following the call instructions (*invoke*) present in the code; and Control Flow Graphs, which are created considering also the jumps in the code caused by conditional and loop statements (*if, switch, for, while*...). In both types of graphs, a node represents a method or a block of instructions that can only be executed sequentially, i.e., a basic block; and the edges represent the execution flow between nodes [183].

[2] `https://github.com/tracer0tong/axmlprinter`
[3] `https://github.com/JesusFreke/smali`

	wallpaper_dev	http://m4lw4.re	0x3d93cb	SUBSCRIBE	getInstalledApps	openConnection	getSmsDetails	getDeviceId	setWallpaper	getLine1Number	getConnectionInfo	exec	SET_WALLPAPER	INTERNET	SEND_SMS	READ_SMS
APP1	0	0	0	1	0	0	1	1	1	0	1	0	0	0	1	1
APP2	0	1	1	0	1	1	0	0	0	0	0	1	0	1	0	0
APP3	1	0	1	0	0	0	0	0	1	1	0	0	1	1	0	0

Fig. 3.2: Binary encoding of the strings (in orange), API calls (in black) and permissions (in blue) for three apps

After performing static analysis, the data obtained from APKs is mapped into feature vectors that represent the apps in a structured way, suitable for processing by ML algorithms. Figure 3.2 depicts a binary mapping that represents three apps by means of their strings, API calls and permissions. It uses the values 1 or 0 to denote whether a feature is present in an app's code or not. Other mappings are also possible, for example, frequency encoding accounts for the number of times a feature is present in the app code.

3.2 Supervised Classification for Android Malware Detection

Once the feature vectors for malware and goodware apps are obtained by means of static analysis, supervised classification ML algorithms can be used to build a malware detection classifier with this data. Supervised classification is a popular ML task in which the objective is to learn a mapping or *classifier* $\hat{H} : \mathcal{X} \rightarrow \mathcal{C}$, where $\mathcal{X}$ is a space of features that describes the samples (the input), and $\mathcal{C}$ is the space of class labels (the output). To do so, the ML algorithm is fed with a set of labeled samples $D = \{(\boldsymbol{x^1}, c^1), ..., (\boldsymbol{x^n}, c^n)\}$ called the training set.

In the context of Android malware detection using ML classifiers, the training set consists of a set of labeled Android apps. Each app or sample is described as a vector of t features $\boldsymbol{x^i} = (x_1, x_2, ..., x_t)$ which, in the context of this dissertation, are computed through the static analysis of its APK file. The binary class label c^i takes a value of 0 for goodware apps and a value of 1 for malware apps. Once the classifier is trained, given the feature vector of a new app $\boldsymbol{x^k}$, it will return its predicted class label $\hat{c}^k$ (0 for goodware and 1 for malware). This is what we call throughout all this work a ML-based Android malware detector.

3.3 App Obfuscation in Android

The performance of malware detectors is highly dependent on the information provided by the collection of features used to represent samples (apps). Accordingly, malware authors employ obfuscation as a means to evade detection by obstructing static analysis, rendering most features ineffective for ML classifiers used in malware detection.

Obfuscation is the process of modifying an executable without altering its functionality [322]. It aims to counteract automatic or manual code analysis. In the Android context, many strategies can be applied to modify the code or resources within the APK file: from simple operations that change some metadata to bypass basic checks (e.g., signature-based anti-malware), to techniques that explicitly modify the DEX code or resources of the app [328]. It is worth emphasizing that obfuscation in Android is more common than in other binary code (e.g., x86 executables), because analyzing and repackaging an Android app is straightforward [253]. The rest of this Section describes the most common modifications.

A. Renaming

As we mentioned, the DEX file of an APK stores the compiled code of an app, including the original string-valued identifiers (names) of fields, methods and classes [113]. Often, these identifiers leak information about code functionalities, lifecycle components and how they interact with each other. For instance, a common practice by programmers is to add "Activity" to each Java class that implements an activity component. The renaming technique replaces these identifiers with meaningless strings, aiming to remove information about the functionality of the app. Consequently, renaming involves modifying the .dex files and the Manifest file (*AndroidManifest.xml*). Note that this technique cannot be applied to methods of the Android lifecycle (e.g., onCreate, onPause) or Android framework components because that would break the execution logic.

B. Code manipulation

These techniques manipulate the DEX code to insert useless operations, hide specific API invocations, and modify the execution flow. The main techniques in this category are:

- *Junk code insertion (JCI)* This technique introduces sequences of useless instructions, such as *nop* (i.e., *no-operation* instructions that do nothing). Other JCI strategies transform the control-flow graph (CFG) of apps by inserting *goto* instructions or arithmetic branches. For example, a *goto* may be introduced in the code pointing to an useless code sequence ending on another *goto* instruction, which points to the instruction after the first *goto*. The arithmetic branch technique inserts a set of arithmetic computations followed by branch instruction that depends on the result of these computations, crafted in such a way that the branch is never taken [17].

- *Call indirection (CI)* This technique aims to modify the call graph and therefore, the CFG of the app. It introduces a new intermediate chain of method invocations in the code, adding one or several nodes between a pair of nodes in the original graph. For example, given a method invocation from m_{or1} to m_{or2} in the code, m_{or1} is modified to call to the start of a sequence of n intermediate methods ($m_i : 1 <= i <= n$) that end in a call to m_{or2}. In this way, the analysis could not reveal that m_{or2} is actually invoked by m_{or1} [246].
- *Reflection* This technique uses the reflection capability of the Java language to replace direct method invocations with Java reflection methods that use class and method identifiers as parameters to perform the call. This makes actual method invocations difficult to inspect [246]. Listings 3.1 and 3.2 show an example of this transformation. In Listing 3.1, the method *m1* (of the class *MyObject*) is accessed through the operator "." from the object instance, whereas Listing 3.2 shows the same invoked method using the Java reflection API. In this example, a *java.lang.reflect.Method.invoke()* object is created (lines 2-3) and invoked (line 4) for a specific object instance (i.e., *obj*), whereas the class and method names are passed as parameters of these functions.

Listing (3.1) Java direct method invocation

```
1   public class com.example.MyObject implements
        MyInterface {
2       public Object m1(Object prm) {}
3   }
4
5   public Object
        standardInvoke(com.example.MyInterface obj,
        Object arg) {
6       return obj.m1(param);
7   }
```

Listing (3.2) Java reflective method invocation

```
1   public Object
        refectionInvoke(com.example.MyInterface
        obj, Object arg) {
2       Class<?> cls =
            Class.forName("com.example.MyObject");
3       Method m = cls.getDeclaredMethod("m1");
4       return m.invoke(obj, param);
5   }
```

C. Encryption

This technique prevents accessing to parts or the entire code or resources (e.g., strings and asset files) of the app by using symmetric encryption algorithms. It involves storing the original code or resources in an encrypted form so that a decryption routine, inserted in the code, is invoked whenever an encrypted part needs

to be accessed. The decryption key is stored somewhere in the APK or calculated at runtime. This technique introduces extra latency during app execution and severely complicates the analysis of the functionality of the encrypted part [328].

Listing (3.3) Java reflective method invocation with encrypted values

```
1  public Object
       refEncrInvoke(com.example.MyInterface
       obj, Object arg) {
2      String className =
           decrypt("AXubduuiao...ZXW");
3      String methodName =
           decrypt("uibdadBUID...ncu");
4      Class<?> cls = Class.forName(className);
5      Method m =
           cls.getDeclaredMethod(methodName);
6      return m.invoke(obj, param);
7  }
```

It is worth emphasizing that different obfuscation techniques can be combined to improve their effectiveness. For example, encrypting the strings of reflective calls can hide the method and class names invoked at runtime. This makes it difficult to recover these values by static analysis of the apps. Listing 3.3 shows an example of the application of both obfuscation techniques to the code in Listing 3.1. In particular, the class and method names are decrypted at runtime (lines 2-3), hiding which methods are actually invoked. Note how these values are exposed only in an encrypted form, and could change if a different encryption key or algorithm was employed.

4

Toward realistic AI-based Android malware detection

As in other areas of cybersecurity research, such as in the network intrusion detection works described in the first part of this thesis, AI techniques have emerged as promising solutions for detecting Android malware. In this regard, many proposals have been presented to date that employ a variety of algorithms and feature sets to create Android malware detectors. The reported results are, in some cases, impressive. However, the lack of reproducibility and the absence of a standard evaluation framework make it difficult to compare these proposals. In this chapter, we analyze ten influential research papers on Android malware detection using a common evaluation framework.

We identify five factors that, if not taken into account when creating datasets and designing detectors, significantly affect the trained ML models and their performances. In particular, we will analyze the effect of (1) the presence of duplicated samples, (2) label (goodware/greyware/malware) attribution, (3) class imbalance, (4) the presence of apps that use evasion techniques and, (5) the time-changing nature of apps. Based on extensive experimentation, we can conclude that the ML-based detectors presented in the literature have been evaluated optimistically, which justifies the good published performance. Our findings also highlight that it is imperative to generate realistic experimental scenarios, taking into account the aforementioned factors, to foster the rise of better ML-based Android malware detection solutions.

4.1 Introduction

Over the past decade we have witnessed impressive advances in mobile devices. Along with the hardware, operating systems (OS) designed for the mobile market have experienced pairwise functionality improvements. With a market share near 72% as of the last quarter of 2021 [277], the Android platform is the leading mobile OS. It has an open source nature and is available for multiple processor architectures. These facts, along with the availability of a well documented development framework that enables a rich set of services (voice and image recognition,

contactless payments, etc.), have contributed to the adoption of Android beyond smartphones [8, 194, 78].

Coupled with this growing popularity, the increasing attention paid by malware writers to this OS has highlighted the risks to which users are exposed [311]. Data stored on mobile devices is of vital importance, sensitive for users, and has become a valuable target for attackers. According to Kaspersky, adware and banking malware targeting these devices were two of the main security threats in 2020, even being detected in trusted app markets such as Google Play [174].

Aware of these problems, researchers have seen AI techniques, and more specifically ML algorithms, as a promising solution for the implementation of Android malware detectors [282]. ML methods leverage on app data to identify signals that are useful for detecting malware. To this end, malware detection can follow one of these strategies (or both in combination): (1) *anomaly-based* detection focuses on building profiles from goodware so that deviations from those profiles are flagged as dangerous; (2) *misuse-based* detection, instead, focuses on learning the characteristics of both malware and goodware in order to identify their presence in new apps [207]. Most of the Android malware detection research belongs to the second group, relying on supervised ML algorithms [88, 281].

To perform the detection of malware using ML algorithms (either for misuse or anomaly-based systems), apps need to be preprocessed in order to extract the set of features that best describe their behavior. This task is performed using dynamic or static software analysis techniques. In *dynamic analysis*, the behavior of an app is monitored in a controlled environment (sandbox), where user and system interactions are simulated. *Static analysis* is based on the inspection of the files contained in the APK without needing to run the code. These techniques have their own advantages and drawbacks. On the one hand, through dynamic analysis, it is possible to access the code that is loaded and executed in runtime. However, the success of this analysis, in terms of coverage of code, greatly depends on the simulation mechanism and the absence of sandbox evasion instruments in the apps [1]. On the other hand, as we previously stated in Chapter 3, static analysis is able to evaluate all the information present in the APKs, but its success lies in the absence of evasion techniques such as obfuscation or dynamic loading of code [210]. Both analysis approaches are complementary and can be combined for Android malware detection [275, 25].

One of the main difficulties faced by researchers when proposing, developing and testing Android malware detectors is the absence of a common and realistic evaluation framework. This framework should include appropriate and labeled datasets, essential for training and/or testing ML algorithms. However, the literature shows that authors opt for building ad-hoc, custom datasets by downloading app samples from different sources and labeling them using tools such as VirusTotal[1]. This process is not only expensive, but also complicates the reproducibility and comparison of ML-based malware detection proposals.

[1] `https://www.virustotal.com`

The issue of reproducibility is aggravated by the unavailability of the code that implements the proposed methods, or by the omission in their respective publications of important details that allow their implementation. For methods using ML algorithms that require tuning a large number of parameters in order to perform properly [232], this information is not often provided. The same is true for the evaluation procedures. In many cases, they are not clearly described or are designed assuming very optimistic scenarios [9, 232].

The main objective of this chapter is to perform a fair comparison of Android malware detection proposals already published in the literature, shedding light about their actual effectiveness. Given the vast amount of proposals presented over the years, as well as the absence of common and realistic evaluation criteria, performing a fair comparison of methods is not a straightforward task. We have chosen 10 popular detectors based on static analysis that use different features and ML methods, and compared them under a common evaluation framework. In many cases, a re-implementation of the algorithms used in the detectors has been required due to the lack of the original authors' implementations. The result of this extensive implementation and experimental work is, to the best of our knowledge, the most comprehensive comparative study on Android malware detection methods presented to date. The scientific contributions of this chapter are summarized as follows:

1. We present a number of factors that negatively affect the accuracy of Android malware detectors. In particular, we consider five conditions that are present in real life but are often ignored when proposing malware detectors: (1) datasets contain a large amount of apps that are almost identical to others; (2) there is not always agreement on what is goodware and what is malware, and some apps lack enough consensus to be considered malicious or benign; (3) there is more goodware than malware; (4) malware authors may resort to evasion attempts using obfuscation techniques; and, (5) the time-changing nature (evolution over time) of malware and goodware. Then, we argue that it is imperative to consider all these factors when designing and evaluating Android malware detectors in order to provide realistic performance values.
2. We analyze the performance of state-of-the-art ML-based Android malware detection approaches when the above factors are taken into account. For this purpose, we selected 10 highly influential detectors that make use of static analysis techniques. We show that the outstanding performances provided by the authors of these approaches are unrealistically optimistic due to design and evaluation flaws.
3. We highlight the lack of reproducibility of published work in this area. In this sense, we make the code and datasets used in this comparative work publicly available.

The structure of this chapter is organized as follows. Section 4.2 reviews the most important works related to the topic of this chapter. The state-of-the-art detectors based on supervised ML classifiers that are considered in this chapter are presented in Section 4.3. Section 4.4 describes how datasets for malware detection

are typically built, their limitations (factors that should be considered when constructing them but are often ignored) and the reproducibility problems that the use of custom datasets entails. In Section 4.5, we present our experimental setup. Section 4.6 discusses the limitations of the selected Android malware detectors based on the analysis of the factors identified in Section 4.4. Section 4.7 discusses the characteristics that a realistic evaluation framework should take into account for Android malware detection. We conclude this chapter in Section 4.8.

4.2 Related Work

Android malware detection is a well-studied area in the information security literature. Despite this, only a few experimental studies have focused on analyzing what factors impact the performance of malware detectors, which is the main theme of this chapter. Most of them focus their analysis on a small group of detectors with similar characteristics. To date, the most comprehensive study is [232]. This article analyzed two different sources of bias in the evaluations of three detection algorithms. The first, known as spatial bias, comes from the differences between the proportion of samples from each class in the dataset. The second type, temporal bias, is related to the inclusion of future knowledge during model training. Experiments to test for spatial bias concluded that the ratio between classes is determinant in the results reported by the authors. Experiments on the impact of temporal bias demonstrated that models tend to misclassify malware as time passes, whereas the accuracy of goodware remains stable over time. The conclusions drawn from this work are similar to those obtained in two other previous studies, each of which considers only one detector in their experiments [251, 9].

The use of obfuscation techniques with malware apps to evade detection was studied in [247]. The work presented an analysis for ten Android commercial antivirus products, testing them by applying different obfuscation techniques to malware. The results proved that all the analyzed tools worsened their effectiveness to detect malware with at least one type of obfuscation. This work served to highlight the weaknesses of these commercial solutions. However, given that the details of the detectors are not public, specific conclusions about how obfuscation may affect ML detectors for Android cannot be drawn from this study. Another work assessed the ability of a ML malware detector for Windows to identify packed[2] malware [4]. An extensive set of experiments was performed to confirm whether packed samples were identified due to the presence of traces left by packers or by the behavior of the sample. The authors concluded that ML detectors relying on static analysis features tend to focus on signs of obfuscation. Thus, putting in question the feasibility of these approaches against Windows malware due to the amount of false positives. Nonetheless, further studies need to be conducted to evaluate if these findings also apply to Android malware detectors.

[2] Developers of malware apps use *packers* to evade detection and analysis. A packer tool compresses and encrypts together data, resources and the code of executable files. These elements are unpacked and executed at run time.

The presence of duplicates in datasets when designing and evaluating detectors is partially discussed in [280]. Preliminary experiments are carried out for two ML detectors, one based on the usage of API calls and one using permissions. The authors of that work postulate towards the existence of a relation between duplicates and the obtention of overestimated performances for models. However, additional analysis would be required to confirm this hypothesis, as the implicit reduction of the size of the dataset for extreme duplicate removal configurations (using ample similarity thresholds) may eventually originate similar results.

In summary, previous works have focused on the analysis of some specific evaluation flaws affecting detectors, including: spatial bias [10, 251, 232], temporal bias [10, 232], the impact of obfuscation [247, 201] or the influence of duplicates in the data [280]. However, these analyses were conducted over a small number of methods [251, 9, 280], in some cases, making use of similar feature sets [232]. Some studies were not deep enough [280] or were performed exclusively for commercial black-box detectors, so that the details about their detection mechanisms, i.e., their features or whether they are based on signatures or ML algorithms, are not disclosed and results cannot be extrapolated [247, 201]. None of these published works has taken into consideration the bias caused due to the removal from datasets of apps that are neither clearly goodware or malware, or has analyzed the effect of using different thresholds to label an app as malware on many existing detectors. Thus, we believe that a more comprehensive analysis, that includes all the identified factors and a higher number of detectors using different features and ML methods, becomes mandatory in the Android malware detection area. This chapter not only provides a comparative framework that evidences the lack of realistic proposals and illustrates many extended design and evaluation biases, but also, gives recommendations towards the proposal of realistic malware detectors.

4.3 Android Malware Detectors

For this analysis we selected 10 malware detection methods for Android. Eight of them are selected because they have been published in in top-tier journals and represent the most important papers in this area of study, in terms of relevance, according to IEEExplore, Scopus and Google Scholar. We have also added two additional proposals, which have been considered in other experimental comparative works, namely the Drebin [24] and BasicBlocks [9]. All of these are *misuse*-based detectors, using different supervised ML classification algorithms and with different sets of features extracted from the apps. In addition, given the large number of features they obtain from APKs, some approaches apply dimensionality reduction algorithms [207].

Table 4.1 outlines the key aspects of these detectors. As can be seen, they were published between 2014 and 2019. Due to the lack of standard datasets, an aspect which is studied in depth in the next section, they all used custom collections of apps to perform their experiments. In many cases, for these proposals, sample selection, class ratios or labeling criteria not only vary, but are not properly documented

Table 4.1: Android malware detection methods included in this analysis. They are the most relevant works according to the literature

Method	Pub. Year	Dataset (#samples)	APK Features	Encoding	Dim. Reduction	ML Algorithm	Reported Performance
AndroDialisys [91]	2017	Google Play[1](1 846) Drebin (5 560)	Permissions Intent Filters	Binary	-	Bayesian Network	TPR: 0.955 FPR: 0.044
BasicBlocks [9]	2016	Google Play[1](52 000) Android Genome (1 260)	Basic Blocks	Binary	Mutual Info	Random Forest	TPR: 0.91 Precision: 0.94
Drebin [24]	2014	Benign[1](123 453) Drebin (5 560)	Permissions App. Components Hw. Components Intents Strings API calls	Binary	-	SVM	TPR: 0.939 FPR: 0.01
DroidDet [334]	2018	Google Play[1](1 065) VirusShare[1](1 065)	Permissions Intents API calls	Binary	tf-idf rank	Rotation Forest	TPR: 0.884 Precision: 0.886
DroidDetector [323]	2016	Google Play[1](20 000) Android Genome (1 260) Contagio[1](500)	Permissions API calls	Binary	Mutual Info[2]	DBN	TPR: 0.981 FPR: 0.201
HMMDetector [57]	2016	Google Play[1](5 560) Drebin (5 560)	Opcodes	Sequence	HMM	Random Forest	TPR: 0.968 Precision: 0.96
ICCDetector [310]	2016	Google Play[1](12 026) Drebin[1](5 264)	Intents App. Components	Binary and frequencies	Mutual Info	SVM	TPR: 0.931 FPR: 0.006
MaMaDroid [222]	2019	Google Play[1](8 447) Drebin (5 560) VirusShare[1](29 933)	API call graph	Frequencies	-	Random Forest	TPR: 0.97 Precision: 0.95
MultimodalDL [164]	2018	Google Play[1](20 000) Android Genome (1 260) VirusShare[1](20 000)	App. Components Intents Hw. Components Permissions Opcodes Strings API calls	Binary	-	DL	TPR: 0.99 Precision: 0.98
PermPair [21]	2019	Benign[1](6 993) Android Genome (1 264) Drebin[1](2 764) Contagio[1](250) Koodous[1](2 975) PwnZen[1](300)	Permissions	Frequencies	Sequential removal	Nearest Neighbors	TPR: 0.951 FPR: 0.042

[1] Unspecified set or labeling criteria from this source.

[2] In the original paper, authors use a filtered set of API calls without stating the criterion used to extract this set. As this information is unavailable, we decided to use Mutual Information to select the most important API calls based on the training data.

also. This makes it difficult to reproduce the experiments, compare proposals or measure their contribution level. Furthermore, none of their authors, excepting those of HMMDetector and MaMaDroid, have made their code publicly available, hindering the possibility to re-run the detectors, and complicating their use for comparison purposes. Because of these, a direct comparison of the performances reported in the corresponding articles does not provide useful information.

Given the unavailability of the code of most of the detectors, we have had to re-implement them in order to perform the experimental analysis contained in this chapter. We tried to reimplement every proposal in the best possible way, so that we can offer fair and accurate comparisons. However, this was a complex task, mainly due to the lack of details concerning crucial aspects such as parameter values, the feature extraction and training processes of the classifier, etc. The most problematic approaches in this sense have been MultimodalDL and PermPair. Unfortunately, we

were unable to implement MultiModalDL [164], due to its complexity and the omission of information regarding feature computations (number of centroids, thresholds for similarity computation). As for PermPair [21], we implemented the detector as indicated in the original publication. However, the results obtained by us are far from those originally reported by the authors. For these reasons, we omit these works from subsequent analysis.

We have implemented the remaining eight malware detectors listed in Table 4.1 in Python language[3]. To obtain the feature sets specific to each detector, we used the *Androguard* framework [75], a widely-used static analysis and reverse engineering tool for Android APK files. For dimensionality reduction, ML algorithms and the assessment of the performance of detectors, we employed well-established libraries such as *scikit-learn* [231] and *numpy* [127]. Our goal is not only to perform a comparative analysis between Android malware detectors, but also to contribute to the reproducibility and progress in the area by releasing our programs and the instructions for their use in our GitLab repository[4].

4.4 Datasets. Drawbacks and Reproducibility Issues

When building a supervised classifier, the dataset and the procedures used for training and evaluation are key factors in order to develop robust and well-performing models [128]. Consequently, for ML-based malware detection, the collection of apps and the process for their generation are particularly important, both in terms of applicability to real-world scenarios and reproducibility. In this section, we identify and discuss a set of factors that must be considered when creating datasets. We then describe the datasets most commonly used by the Android malware detection research community, pointing out their main drawbacks from the point of view of the identified factors.

4.4.1 Factors Under Analysis

We have identified five factors that have a major impact on the performance of ML based malware detectors for Android, and that should be considered when creating the datasets used to train and evaluate them. It should be noted that, in our opinion, these factors are very important, but this does not imply that they are the only influential aspects in the design and evaluation of malware detectors based on supervised classification.

4.4.1.1 REDUNDANCY

The purpose of Android malware detectors is to identify every type of malware regardless of their level of incidence. This includes a wide range of samples pertaining

[3] In the case of DroidDetector [323], we were able to implement the detector assuming that feature selection was used to identify the set of most relevant API calls used by the algorithm.

[4] `https://gitlab.com/serralba/androidmaldet_comparative`

to different malware families and subfamilies. Typically, malware samples within a family or subfamily exhibit analogous code and data structures to perform the same malicious activities [299]. This characteristic of malware results in datasets with groups of samples that are very similar from the point of view of detectors. This means that, within these groups, samples tend to be represented using identical feature sets, resulting in redundancies in the data fed to ML models. On the one hand, redundant samples in the training set cause bias in ML algorithms because decisions towards groups with many representatives have a great impact in the accuracy of the model [128]. This makes models to ignore non-redundant samples, since predictions for these instances yield little improvement in accuracy. Resulting, thus, on detectors with a limited ability to identify uncommon (less represented) forms of malware. On the other hand, the presence of duplicates in the evaluation set leads to inflated or poor performances, depending on whether or not these large groups of similar apps are correctly classified. Even if the presence of duplicates or very similar apps has an important effect on the results obtained by a detector, to the best of our knowledge, only preliminary work has considered it as an issue [280]. As a matter of fact, none of the detectors considered in our comparative analysis were assessed taking into account redundancies in the data as an major source of bias.

4.4.1.2 LABELING-GREYWARE

One of the main problems when building datasets for Android malware detectors based on supervised classification lies in the need to have labeled samples. The true nature of an app is not known in many cases. This implies that apps need to be analyzed in order to assign them a label concerning their maliciousness. Ideally, labeling should be carried out manually, by experts, to guarantee the highest number of error-free labels. However, using human annotators to perform this task is costly, both in terms of time and resources [184] and it is not error-free either. Accordingly, researchers usually rely upon automatic procedures to label their customized datasets.

In the simplest labeling approach, apps are labeled according to the source they were obtained from [21], e.g., those downloaded from trusted repositories (such as Google Play) are goodware, whereas those from repositories such as VirusShare are labeled as malware. This procedure relies on the analysis, either automatic or manual, performed by the managers of these repositories. In other cases, apps are scanned using a collection of antivirus programs and labeled depending on the number of positive (malware) alerts [273]. In this context, VirusTotal is a tool that allows users to upload files, including APKs, and scan them using a collection of antivirus engines. VirusTotal results, based on the number of positive alerts raised by a file (we refer to this number as VTD, from VirusTotal Detections), are widely used as the criterion to label samples. To this end, a common procedure makes use of thresholds to establish the level of consensus required to label an APK as malware or goodware [255].

Leveraging on the VTD leads authors to decide what threshold is adequate for malware and goodware, so it is common to find disparities in the literature. For example, [251] set a threshold of VTD$\geq$10 to flag an app as malware, while in [232], authors set this threshold at a much lower value of 4. In both cases, the condition for labeling an app as goodware was set to VTD=0. The choice of thresholds not only means that the malware or goodware definitions vary among articles, but also influences the amount of apps that fall between these categories. These apps, which we refer to as greyware, lack enough consensus by antivirus programs to be considered as goodware or malware with guarantees. Because of that, many researchers discard these apps when training their detectors. However, greyware is an important part of the Android ecosystem [111] and will appear whenever the detector is deployed in a real environment. Thus, discarding them at training time will hinder the effectiveness of a detector once deployed. Indeed, even if common thresholds were applied to label the data, the VTD value provided by VirusTotal changes over time [335], for example, due to engine updates aimed at improving detection capability, or as engines are added or removed from the platform. Consequently, disparities may occur between models validated with exactly the same collection of apps but labeled at different times [255].

4.4.1.3 IMBALANCE

The third factor we consider is related to the ratio of malware and goodware in the dataset, especially in the data used for training. In real life, most apps are innocuous in the security aspect, with the actual proportion of malware being about 10% of the total number of Android apps [232], but with this ratio being highly dependent on the particular market from which apps are downloaded [112, 187]. As can be seen in Table 4.1, researchers have trained their detectors following their own criterion and assuming different class proportions. However, the choice of the class ratio is important when training ML models, since classical ML classifiers tend to be biased towards the majority class in highly unbalanced scenarios. Therefore, ignoring class imbalance can lead to a false perception about the true performance of detectors under real working conditions [232].

4.4.1.4 EVASION

Some malware authors are aware that their apps will be examined by malware detectors, so they try to bypass detection by using different evasion techniques. As technology evolves, different threats may appear and authors should be aware that their systems will be the target of attacks. Thus, a proper evaluation of the robustness of malware detectors against such threats should be taken into consideration. Unfortunately, none of the proposals included in this comparative work have considered the effects of evasion techniques, such as obfuscation, in their evaluations.

4.4.1.5 EVOLUTION

The last factor we highlight is related to the time-changing nature of malware and goodware. This implies that the behavior of apps rarely remains static for a long time. Thus, the characteristics of newer or mutated apps may differ from those obtained from apps observed earlier, during the training of a detector. Despite that, most authors design and/or evaluate their ML detectors on the assumption that future malware and goodware will remain similar to that used at design time [251, 232].

4.4.2 Available Android Datasets and their Drawbacks

Android datasets that are used for building supervised classifiers consist of collections of apps and their associated labels, which indicate whether an app is malware or goodware. We have searched in the literature for datasets of Android APKs designed for research on misuse-based Android malware detectors, and have found five popular ones. Their characteristics have been summarized in Table 4.2, taking into account all the factors identified in the previous section.

Table 4.2: Characteristics of popular Android Malware Datasets. The "?" symbol means that no information about this characteristic is reported

Dataset	Time period	Labeling method (Type of labels)	#samples	Obfuscated samples	Redundant samples	Timestamps
Android Genome [333]	2010-2011	Manual (Binary)	1 260 malware	?	?	✗
Drebin [24]	2010-2012	VirusTotal (Binary)	5 560 malware	?	?	✗
AMD [299]	2010-2016	Hybrid (Binary)	24 553 malware	?	?	✗
CICAndMal2017 [177]	2014-2017	VirusTotal (Binary)	426 malware 5 065 goodware	?	?	✗
AndroZoo [11][1]	2011-	VirusTotal (VTD)	13 045 285 mixed	?	?	✓

[1] This dataset is continuously growing

In two of the reported datasets (Drebin and CICAndMal2017), the labels of the samples were obtained by setting some threshold over the VTD. For example, in Drebin, an app is tagged as malware when its VTD$\geq$2, for a subset of eight antivirus engines from VirusTotal that are selected as reliable by the authors. Only the Android Genome dataset was built based on manual labeling. A combination of both labeling approaches was used in the AMD collection: automatic labeling was first carried out using VirusTotal to filter and cluster apps into malware families, and then a small subset from each family was manually verified. Finally, note that AndroZoo does not provide labels, supplying VTD values instead –so it is up to the user how to use this information for labeling.

To properly train detectors based on ML classifiers, both malware and goodware samples are needed. Ideally, greyware should also be included. Nonetheless, Drebin, Android Genome and AMD comprise exclusively malware samples and only AndroZoo allows samples to be labeled as greyware. Another drawback of these datasets

is related to obfuscated malware. In this sense, the authors neither identify, or explicitly include, obfuscated versions of malware, which makes it very difficult to analyze the possible effects of evasion attempts on the performance of detectors. Something similar occurs for redundant samples, as none of the papers describing the datasets provide information about their presence.

Finally, as can be seen in Table 4.2, most of these datasets were created more than five years ago and may contain old-fashioned malware. The most recent dataset is CICAndMal2017, which comprises a small set of apps released between 2014 and 2017. This small number of instances may not be enough for training and evaluating anti-malware methods based on some ML algorithms [249]. In addition, in most cases the samples included in these datasets cover only a small period of time or their release date is not available. These facts complicate the elaboration of proper analysis about the evolution of the characteristics of apps.

The limitations of these datasets have led researchers to build custom datasets, in a similar way as AndroZoo was created, i.e., by combining APKs downloaded from various sources or repositories. AndroZoo constitutes the most important public source of Android apps for researchers [11]. Sources of AndroZoo include app marketplaces (such as Google Play), malware datasets, torrents and different malware repositories[5]. Since 2011, these sources are continuously tracked to keep the collection up to date. Additionally to the APKs, AndroZoo provides a file with information about the apps contained in the dataset. The contents of this file, which is updated daily, can be used to filter the samples to be downloaded according to different criteria, such as the market from which an APK was obtained, the SHA and the date of the APK, and the VTD value.

From the above discussion, it is clear that the decisions made by the authors during the construction of custom datasets make them unique, and that particularities of these datasets influence the performance of detectors. Moreover, performance is not only dependent on the data used, but also on the design of the experiments using these data. In our opinion, and as mentioned in the previous section, the use of different datasets and the consideration of different guidelines in the design of the experiments, makes the comparison of the metrics reported for these proposals meaningless, entails an obstacle to the reproducibility and seriously impairs progress on this relevant topic.

4.5 Experimental Setup

In this section we present the experimental setup devised for this chapter. We describe the datasets used, the process to build and evaluate the Android malware detectors and the evaluation metrics considered to measure their performance.

[5] For example: ContagioDump (`https://contagiodump.blogspot.com/`), Koodous (`https://koodous.com`), VirusShare (`https://virusshare.com/`).

4.5.1 Master Dataset

Our starting point is a "master" dataset that allows us to derive datasets useful to evaluate on the selected detectors every factor presented in Section 4.4.1, i.e., REDUNDANCY, LABELING-GREYWARE, IMBALANCE, EVASION and EVOLUTION.

To build our master dataset we selected apps from AndroZoo, taking into account their origin, the reported VTD and the date. We use apps from Google Play, AppChina and VirusShare. The date of the apps was used to select 100 monthly samples of each class (malware, goodware and greyware) during the period starting from January 2012 to December 2019. As we mentioned earlier, there is a lack of ground truth in the area, and automated labeling methods based on VirusTotal have become commonplace. Given the lack of standard criteria in the literature to interpret the results obtained from VirusTotal [255], in this chapter we resort to a simplified but operational automated labeling method: the use of VTD values and the application of a set of pre-defined thresholds. With this method we have a practical definition of what is goodware, what is greyware and what is malware that allows us to fairly compare detectors and to show different sources of experimental bias. In particular, we used the median of the thresholds used in two previous works [251, 232] to label the malware, i.e., a VTD≥7. Goodware samples were considered as those with VTD=0, whereas samples with a 1≤VTD≤6 rating were labeled as greyware. Moreover, we restricted our analysis to samples discovered at most in 2019 to ensure that malware signatures for samples in our dataset are well stabilized. Such criteria are in concordance with what is recommended in [335] to obtain reliable labels from VirusTotal.

In total, our master dataset consists of 28,800 samples. The complete list of APKs and the instructions to download this dataset, along with our code, are available in our GitLab repository. In contrast to the datasets introduced in Section 4.4.2, this dataset is more comprehensive as: (1) it considers greyware; (2) it provides a sufficiently large number of goodware, malware and greyware samples to properly train and test ML detectors; (3) it considers a larger time period and the organization of samples by months allows us to split our dataset according to the age of the apps. Additionally, compared to the datasets used for training the detectors included in this analysis (see Table 4.1), the procedure followed to build our master dataset and its derived datasets is more systematic, rigorous and transparent.

4.5.2 Model Training and Assessment Process

For model construction and evaluation, the dataset is divided into training and test partitions. In order to obtain unbiased results, the test partition is always kept as a completely separate set and is never used for training nor for feature engineering processes (extraction, preprocessing or dimensionality reduction). The model parameters are selected using standard k-fold cross validation (with $k = 5$) within

Table 4.3: Evaluation metrics used in this chapter

$TPR = \frac{TP}{P}$	$FPR = \frac{FP}{N}$	$Precision = \frac{TP}{TP+FP}$
$F1 = 2 * \frac{TPR*Precision}{TPR+Precision}$		$A_{mean} = \frac{TPR+1-FPR}{2}$
$kappa = \frac{P_o - P_c}{1-P_c}$		$P_o = \frac{TP+TN}{P+N}$ $P_c = \sum_k p_k * \hat{p}_k$

the training set and following a grid search approach. In scenarios where EVOLUTION is considered because the distributions of the data are assumed to change over time, time-aware k-folds are used [20], i.e., all the apps of the evaluation sets are more recent than those for training. In all cases, folds are created maintaining the original ratio between the classes in the given experimental scenario. This is a common process in the ML literature for estimating the generalization error of the final model [128].

4.5.3 Evaluation Metrics

We considered a set of evaluation metrics (see Table 4.3) which are common in the ML and computer security literature [207]. The TPR, the FPR or precision make use of: true positives (TP), which indicate the number of correct positive predictions; false positives (FP), that account for the number of incorrectly predicted negative elements; and P and N, which make reference to the number of positive and negative elements in the data. F1 represents the harmonic mean between precision and TPR, A_{mean} is the arithmetic mean of the accuracies for the positive and negative classes, and the kappa statistic quantifies the level of correlation between the predictions made by a classifier and the actual labels in the data (P_o being the accuracy of the detector, P_c being the weighted sum of the predictions, p_k being the actual proportion of samples of the class k in the data and $\hat{p_k}$ being the proportion of samples predicted as pertaining to class k). Unlike the F1 score (as it does not consider True Negatives), the kappa and A_{mean} metrics are especially useful under unbalanced problems.

4.6 Comparative Analysis

In this section we run a complete set of experiments, designing specific scenarios to analyze the effect of the factors presented in Section 4.4.1. First, a basic scenario is used as the departing point for subsequent experiments. In all scenarios, the selected detectors are compared in equal conditions.

Table 4.4: Composition of Balanced and Unbalanced datasets derived from the 2012-2015 period of the Master Dataset

	Training			**Testing**		
	%malware (#samples)	**%goodware (#samples)**	**ratio**	**%malware (#samples)**	**%goodware (#samples)**	**ratio**
Balanced	70% (3360)	70% (3360)	1:1	30% (1440)	30% (1440)	1:1
Unbalanced	∼7% (∼336)	70% (3360)	∼1:10	∼3% (∼144)	30% (1440)	∼1:10

Departing from the master dataset, two different datasets have been extracted, namely balanced and unbalanced (see Table 4.4). For the balanced case, it is assumed that malware and goodware are equally probable. Thus, from the period 2012-2015, ∼70% of all the goodware and ∼70% of all the malware are uniformly sampled on a monthly basis and used for training, keeping the remaining 30% of the samples for testing purposes. On the contrary, under the unbalanced configuration, malware is assumed to be less frequent than goodware and, consequently, the malware is downsampled to become ∼10% of the goodware. This process is performed by randomly sampling malware on a monthly basis according to a Gaussian distribution with parameters $N(0.1, 0.02)$.

4.6.1 Baseline Scenario

Before analyzing the scenarios related to each factor, we test detectors under the most basic and unrealistic assumptions, i.e., discarding greyware, considering a balanced ratio between the classes, omitting evasion attempts and ignoring the evolution of the apps. The aim of this scenario is to mimic the (favorable) conditions that are commonly assumed in the literature and try to reproduce the results obtained in the original papers.

As shown in Table 4.5, half of the detectors showed promising detection performances in this scenario, achieving detection rates (TPR) over 0.8 and moderately low false positives of about 0.1. The best detector under this scenario in terms of summary metrics (0.91 for kappa), despite being one of the first works in the area, was Drebin, with TPR and FPR values of 0.95 and 0.04, respectively. Another two well-performing detectors were DroidDet and MaMaDroid, with similar kappa figures of 0.84, despite using different features and ML algorithms. The good results reported for some methods contrast with those obtained for DroidDetector, HMMDetector and AndroDialysis. Given the high number of false positives of these detectors (above 20%), one could conclude that these models are inappropriate even under optimistic working conditions.

4.6.2 Redundancy Scenario

This scenario aims to study the impact of the presence of very similar samples in Android datasets. To this end, we filter out malware and goodware apps from our

Table 4.5: Performance of detectors for the baseline scenario

Method	TPR	FPR	Precision	F1	A_{mean}	Kappa
AndroDialysis	0.848	0.291	0.744	0.792	0.778	0.556
BasicBlocks	0.823	0.097	0.894	0.857	0.863	0.726
Drebin	0.953	0.043	0.956	0.955	0.955	0.910
DroidDet	0.936	0.088	0.913	0.925	0.924	0.848
DroidDetector	0.472	0.335	0.585	0.523	0.568	0.137
HMMDetector	0.824	0.560	0.595	0.691	0.631	0.263
ICCDetector	0.800	0.093	0.895	0.845	0.853	0.706
MaMaDroid	0.930	0.085	0.915	0.923	0.922	0.845

initial balanced dataset. This is done by computing intra-class similarities between samples and then applying the algorithm proposed in [280]. The algorithm works by randomly selecting one sample at a time and removing the samples lying in its neighborhood, according to an ϵ radius threshold.

For our experiments, we use call frequencies of APIs as the representation of apps and the Euclidean distance to compute the degree of similarity between samples. The intuition behind the consideration of this representation lies in that API call frequencies make reference to both the set of API calls that describe the actions carried out by apps, and the prevalence of these calls in the code. Thus, we assume that when the Euclidean distance for two slightly different apps is below or equal to ϵ, both apps perform identical actions, e.g., they are variations of the same app.

We perform the filtering process using a redundancy tolerance value ϵ equal to 0, which means that we consider as duplicates two apps with *exactly* the same API call frequencies. As a result of this process, a dataset without exact duplicates is obtained. A summary of the characteristics of this dataset are presented in Table 4.6. As can be seen, the size of the malware subset is reduced substantially (almost half of the malware was filtered out), whereas for goodware the number of removed duplicates is considerably smaller. Details about the groups found for malware during the deduplication process are shown in Figure 4.1. It depicts the number of groups found in the malware dataset arranged by group size. As can be seen, our filtered dataset is heterogeneous in terms of unique malware behaviors. More specifically, 2285 samples only contain a duplicate, which represent 88% of the filtered malware. Groups with a small number of duplicates represent the majority of the data, with 70% of the original malware containing less than 10 identical samples. Large groups are also present. In this regard, about 2% of the resultant malware groups have more than 10 duplicates.

In order to confirm the influence of the presence of very similar apps in the generalization ability of ML models, the results for detectors trained with filtered data[6] and evaluated with the unfiltered test set used for the baseline scenario are shown in Table 4.7. As can be seen, almost all approaches improve their performances, with

[6] As in the baseline scenario, a balanced ratio between goodware and malware is used for training. This is achieved by randomly downsampling the majority class (goodware).

Table 4.6: Composition of the full dataset used for the filtered scenario (data from 2012 to 2015)

Dataset	**#malware**	**#goodware**
Original	4800	4800
Filtered	2588	4291

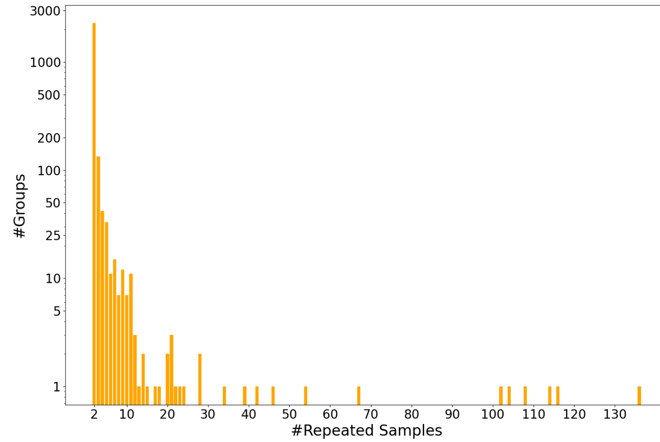


Fig. 4.1: Number of malware groups per group size. Note the logarithmic scale in the Y axis.

Table 4.7: Performance of detectors trained using the filtered dataset and evaluated with the baseline test set

Method	**TPR**	**FPR**	**Precision**	**F1**	$\mathbf{A}_{mean}$	**Kappa**
AndroDialysis	0.904	0.365	0.712	0.796	0.769	0.538
BasicBlocks	0.893	0.044	0.952	0.922	0.924	0.849
Drebin	0.933	0.031	0.966	0.949	0.950	0.901
DroidDet	0.933	0.056	0.942	0.937	0.938	0.876
DroidDetector	0.654	0.403	0.618	0.635	0.625	0.250
HMMDetector	0.796	0.488	0.619	0.697	0.653	0.307
ICCDetector	0.800	0.053	0.937	0.863	0.873	0.747
MaMaDroid	0.914	0.046	0.951	0.932	0.934	0.868

similar TPR and lower FPR values, compared to the results for detectors trained with the unfiltered training set (see Table 4.5). Among the most benefited methods in terms of performance are ICCDetector and BasicBlocks, with improvements of 5% and 16% in their kappa values, respectively. The results observed for this scenario support the hypothesis that duplicate samples in the training set result in biased models, since accurate predictions for these large groups at training time result in higher accuracy, and the models are less prone to pay attention to minority

Table 4.8: Performance of detectors trained and evaluated with filtered sets

Method	**TPR**	**FPR**	**Precision**	**F1**	$\mathbf{A}_{mean}$	**Kappa**
AndroDialysis	0.913	0.362	0.716	0.802	0.775	0.551
BasicBlocks	0.880	0.065	0.930	0.904	0.907	0.814
Drebin	0.925	0.056	0.942	0.933	0.934	0.868
DroidDet	0.926	0.073	0.926	0.926	0.926	0.853
DroidDetector	0.728	0.395	0.647	0.685	0.666	0.332
HMMDetector	0.774	0.545	0.586	0.667	0.614	0.229
ICCDetector	0.797	0.070	0.918	0.853	0.863	0.726
MaMaDroid	0.900	0.072	0.925	0.913	0.914	0.828

groups. Thus, removing duplicates from training is translated into models that are able to generalize better to different types of instances.

The previous experiment was carried out with an unfiltered test set. However, to evidence the impact that the presence of duplicates in the testing set has on the reported performances, we perform evaluations using a filtered test set. Table 4.8, depicts the performance for detectors both trained and evaluated using data without duplicates. Among all the methods, the best scoring are Drebin and DroidDet with kappa values of 0.86 and 0.85, respectively. However, as expected, most detectors report lower detection performances in this scenario than in the previous experiments (see Table 4.7 and Table 4.5). With duplicates in the test set, correctly identifying larger groups has a significant positive effect on the measured performance, whereas erroneously classifying these groups penalizes it. This effect disappears when duplicates are removed and only one representative per group is left. Therefore, the measured performance is a better indicator of generalization ability of detectors.

4.6.3 Labeling-Greyware Scenarios

As previously mentioned, manual, exhaustive and error-free labeling is not affordable. For this reason, the community has agreed to use tools such as VirusTotal to automatically label apps. In this sense, using thresholds on the VTD value is the most common class separation criteria. However, different interpretations and thresholds have been used, which impacts on the obtained datasets. Two interesting analyses arise here: (1) how the selection of the threshold affects performance and, (2) what is the behavior of detectors when they examine apps that were discarded during labeling.

First, we evaluate how the selection of different criteria for labeling datasets affects detectors and thus, the reported performance. Specifically, we want to analyze whether the uncertainty and difficulty of the problem is greatly reduced when higher VTD values are used for labeling malware. To do so, models are trained similarly to the baseline scenario, i.e., using a balanced ratio between both classes with samples selected from 2012 to 2015, but varying the threshold used for labeling.

Results for this experiment are shown in Table 4.9. As can be seen, the TPR of models is directly correlated with the VTD used for labeling the dataset i.e., the higher the VTD, the higher the TPR of the detector. The contrary is true for the FPR, since it follows a reverse trend with respect to the VTD. This confirms that the malware detection problem becomes easier whenever the separation between both classes, in terms of the VTD, is enlarged. Figure 4.2 exhibits this effect for the values of A_{mean} obtained for the different detectors.

The second scenario is devoted to show how detectors behave when facing greyware. Specifically, we want to analyze whether the uncertainty and difficulty of the problem increases when adding greyware and, thus, the omission of these apps results on the oversimplification of the problem and leads to unrealistic and unfair evaluations. To do so, models are trained using the balanced dataset and tested exclusively on greyware samples from the period between 2012 and 2015.

We have compiled the responses of the detectors when classifying greyware samples in Table 4.10. G indicates how many input samples are identified as goodware, whereas M refers to how many samples are classified as malware. The total results are in the right column of the table. The other columns show the partial results for the input samples grouped by their VTD scores. As can be seen, on average 35% of the total samples are considered goodware by detectors. As expected, the results show certain correlation between the VTD and the decisions made by the detectors: given an app, the higher its VTD is, the more likely the detectors are to classify it as malware. We can observe a high uncertainty for samples with lower VTD values, explaining why authors opt to discard greyware from their experiments. As a result, the problem solved by detectors is simplified, therefore providing artificially boosted performance results and hiding an effect that will appear in real-working conditions.

In addition, we also analyze how the VTD value and thus, the labels, change over time. For the labeling of our dataset we used the VTD scores provided by AndroZoo. We completely reanalyzed the apps in our dataset with VirusTotal and computed the confusion matrix to represent the label swaps between the two analyses. As can be seen in Table 4.11, most swaps occur for the greyware class. In this sense, more than half of the apps change their label from greyware to goodware (27.2%) or malware (28.1%). Also, 9.9% of the goodware and 6.9% of the malware apps fall into the category of greyware after being reanalyzed. These changes illustrate the downsides and limitations of using the VTD score for labeling. On the one hand, the need of providing the labels of samples when releasing datasets to avoid class swaps caused by the reanalysis of apps and to guarantee reproducibility. On the other hand, the need to reconsider greyware as part of the datasets, as 44% of the apps remain in this category even after being reanalyzed.

4.6.4 Imbalance Scenario

According to previous literature, around 10% of apps are actually malware [232]. Similarly, in this scenario we assume that there will be a strong imbalance between malware and goodware. In order to analyze the behavior of the different detectors

Table 4.9: Performance of detectors trained with data labeled using different thresholds over the VTD for the malware. TPR stands for the True Positive Ratio and FPR is the False Positive Ratio

	$VTD \geq 1$		$VTD \geq 2$		$VTD \geq 3$		$VTD \geq 4$		$VTD \geq 5$		$VTD \geq 6$	
Method	**TPR**	**FPR**	**TPR**	**FPR**	**TPR**	**FPR**	**TPR**	**FPR**	**TPR**	**FPR**	**TPR**	**FPR**
AndroDyalisis	0.547	0.258	0.622	0.275	0.549	0.183	0.692	0.305	0.798	0.369	0.861	0.298
BasicBlocks	0.731	0.184	0.869	0.112	0.885	0.107	0.904	0.134	0.909	0.090	0.916	0.083
Drebin	0.731	0.227	0.860	0.113	0.901	0.107	0.914	0.131	0.935	0.081	0.923	0.062
DroidDet	0.752	0.236	0.873	0.153	0.901	0.152	0.904	0.174	0.935	0.154	0.930	0.083
DroidDetector	0.526	0.384	0.826	0.625	0.937	0.813	0.865	0.757	0.751	0.467	0.763	0.506
HMMDetector	0.759	0.600	0.775	0.588	0.804	0.589	0.794	0.536	0.733	0.506	0.784	0.562
ICCDetector	0.612	0.180	0.732	0.120	0.780	0.096	0.797	0.156	0.832	0.141	0.833	0.055
MaMaDroid	0.756	0.199	0.879	0.136	0.923	0.159	0.923	0.187	0.901	0.145	0.895	0.076

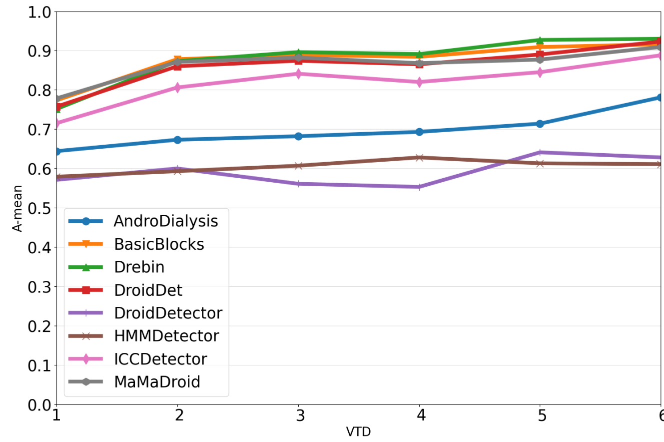


Fig. 4.2: A_{mean} values for different VTD labeling thresholds

in this context, the unbalanced dataset is used. It is worth noting that making the test set balanced (or unbalanced) does not harm the capabilities of the model, but only has an effect in some performance metrics. In this sense, the problem of imbalance in the test set is easily solved by using suitable metrics, such as the Kappa or A_{mean}, that do not conceal the errors for the minority class [207].

Table 4.12 shows the results obtained for detectors under this scenario. The use of unbalanced data for training results in a reduction in the proportion of correctly classified malware (TPR), with respect to the results obtained using the baseline configuration (see Table 4.5). Imbalance in the training data is translated into less malware information provided to the algorithms, making it difficult for detectors to learn the characteristics of this class of samples. On average the performance of most methods decreased about 12% as measured by their A_{mean} values. Among the best scoring methods in the baseline scenario, MaMaDroid is one of the methods

Table 4.10: Decisions made by baseline detectors when dealing with greyware samples from the period 2012-2015, for different VTD values. G stands for the ratio of identifications as goodware, while M is the ratio of identifications as malware

	$VTD = 1$		$VTD = 2$		$VTD = 3$		$VTD = 4$		$VTD = 5$		$VTD = 6$		$1 \leq VTD \leq 6$	
Method	**G**	**M**	**G**	**M**	**G**	**M**	**G**	**M**	**G**	**M**	**G**	**M**	**G**	**M**
AndroDyalisis	0.504	0.496	0.356	0.644	0.272	0.728	0.307	0.693	0.275	0.725	0.200	0.800	0.319	0.681
BasicBlocks	0.675	0.325	0.452	0.548	0.394	0.606	0.333	0.667	0.248	0.752	0.248	0.752	0.392	0.608
Drebin	0.696	0.304	0.460	0.540	0.379	0.621	0.323	0.677	0.251	0.749	0.148	0.852	0.376	0.624
DroidDet	0.700	0.300	0.437	0.563	0.336	0.664	0.297	0.703	0.221	0.779	0.159	0.841	0.358	0.642
DroidDetector	0.629	0.371	0.691	0.309	0.607	0.393	0.659	0.341	0.607	0.393	0.536	0.464	0.622	0.379
HMMDetector	0.415	0.585	0.296	0.704	0.311	0.689	0.259	0.741	0.255	0.745	0.163	0.837	0.283	0.717
ICCDetector	0.683	0.317	0.460	0.540	0.386	0.614	0.381	0.619	0.295	0.705	0.240	0.760	0.408	0.593
MaMaDroid	0.677	0.323	0.423	0.577	0.366	0.634	0.313	0.687	0.217	0.783	0.137	0.863	0.356	0.645

Table 4.11: Label swaps for goodware (G), greyware (X) and malware (M); between annotations of samples in the master dataset using information contained in AndroZoo (rows) and VirusTotal reanalysis reports (columns)

		VirusTotal reanalysis		
		G	**X**	**M**
AndroZoo analysis	**G**	8 563	954	83
	X	2 613	4 284	2 703
	M	17	663	8 920

Table 4.12: Performance of detectors trained with 1:10 malware/goodware ratio

Method	**TPR**	**FPR**	**Precision**	**F1**	$\mathbf{A}_{mean}$	**Kappa**
AndroDialysis	0.373	0.032	0.543	0.442	0.670	0.396
BasicBlocks	0.460	0.017	0.734	0.565	0.721	0.531
Drebin	0.740	0.017	0.816	0.776	0.861	0.754
DroidDet	0.686	0.013	0.837	0.754	0.836	0.731
DroidDetector	0.266	0.162	0.145	0.188	0.552	0.076
HMMDetector	0.106	0.010	0.516	0.176	0.548	0.149
ICCDetector	0.520	0.013	0.795	0.629	0.753	0.599
MaMaDroid	0.513	0.008	0.865	0.644	0.752	0.617

that suffered a more significant decrease in this scenario, with a reduction of 18% in its A_{mean} value.

4.6.5 Evasion Scenario

This scenario is devoted to test the robustness of ML-based malware detectors under conditions where attackers attempt to bypass detection using obfuscation. Our aim with this section is to illustrate the need of carrying out the security analysis of detectors. We focus our evaluation in obfuscation because it is a classical evasion technique that is commonly overlooked by the authors of proposals [124].

In addition to the analyses presented in this section, the effect of obfuscation in detectors leveraging static analysis information will be studied in more detail in Chapter 5.

For this experiment, we selected a set of obfuscation strategies that are commonly used in the wild to hide Android malware behaviors [80]. For each obfuscation strategy considered, as well as for the combination of all of them, a new obfuscated dataset is obtained by: (1) randomly sampling 10% of the apps from 2012 to 2015 in our master dataset and, (2) applying the required transformations to such apps using the ObfusAPK [17] and the AAMO [237] tools. These transformations are:

- Renaming. The original name and identifiers of user-defined classes, fields and methods are changed by meaningless strings.
- Changes in the structure of the code. This form of obfuscation includes: (1) call indirections to add an intermediate function that calls the function originally present in the code, (2) insertion of *goto* instructions, (3) inversion of conditionals to modify the execution flow of the app, (4) insertion of junk code, and (5) reflection[7] to hide function calls to internal code and to the Android framework (APIs).
- Encryption. This technique involves: (1) the generation of a encryption/decryption random key, (2) the encryption of native libraries, strings and assets, and (3) the insertion of a decryption code which is called from every part of the app where these resources are requested.

The results are summarized in Table 4.13. On average, when considered separately, the most successful strategies for evading detection are changes in code structure and encryption. However, generally speaking, individual obfuscation strategies are not as effective as a combination of all of them. Most detectors are prone to misclassify obfuscated malware as goodware, as shown by the decrease in their TPR values for the combined scenario. In this regard, the least affected model is Drebin, being able to identify 79% of the obfuscated malware, with 13% false positives. Others, such as DroidDet and MaMaDroid obtained more moderate performance figures, both identifying around 70% of the obfuscated samples according to their A_{mean} values when using the three obfuscation techniques in combination.

4.6.6 Evolution Scenario

Both malware and goodware evolve over time, i.e., do not follow a stationary distribution. Therefore, it is logical to think that static detectors trained with apps for a certain period of time will not necessarily work well with more recent apps. To prove this assumption, the models trained using the balanced configuration (with data from the period 2012-2015) are tested with goodware and malware obtained between 2016 and 2019.

[7] Reflection is a feature of some programming languages that allows an executing program to examine or "introspect" upon itself, and manipulate internal properties of the program.

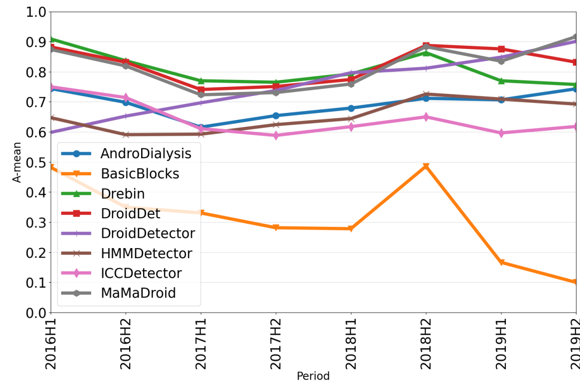
Table 4.13: Performance of detectors with obfuscated apps for the evasion scenario

	Renaming (Rn)			**Changes in code structure (Co)**			**Encryption (Enc)**			**Rn+Co+Enc**		
Method	**TPR**	**FPR**	$\mathbf{A}_{mean}$	**TPR**	**FPR**	$\mathbf{A}_{mean}$	**TPR**	**FPR**	$\mathbf{A}_{mean}$	**TPR**	**FPR**	$\mathbf{A}_{mean}$
AndroDialysis	0.793	0.288	0.752	0.376	0.072	0.652	0.431	0.074	0.678	0.459	0.086	0.686
BasicBlocks	0.624	0.163	0.73	0.211	0.631	0.29	0.418	0.137	0.64	0.172	0.095	0.538
Drebin	0.95	0.012	0.968	0.968	0.025	0.971	0.738	0.138	0.799	0.796	0.136	0.829
DroidDet	0.931	0.028	0.951	0.560	0.024	0.767	0.934	0.021	0.956	0.586	0.029	0.778
DroidDetector	0.154	0.216	0.469	0.049	0.102	0.473	0.133	0.126	0.503	0.157	0.211	0.473
HMMDetector	0.895	0.489	0.702	0.892	0.786	0.553	0.802	0.684	0.558	0.911	0.779	0.565
ICCDetector	0.246	0.033	0.606	0.667	0.023	0.822	0.668	0.023	0.822	0.279	0.024	0.627
MaMaDroid	0.733	0.034	0.848	0.737	0.042	0.845	0.92	0.037	0.941	0.533	0.048	0.743

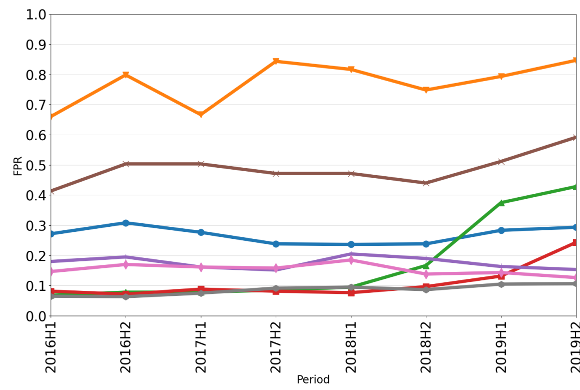
Table 4.14: Performance of baseline detectors using evaluation data between 2016 and 2019

Method	**TPR**	**FPR**	**Precision**	**F1**	$\mathbf{A}_{mean}$	**Kappa**
AndroDialysis	0.656	0.268	0.709	0.682	0.694	0.388
BasicBlocks	0.390	0.771	0.336	0.361	0.309	-0.381
Drebin	0.787	0.171	0.820	0.804	0.808	0.616
DroidDet	0.752	0.108	0.873	0.808	0.821	0.643
DroidDetector	0.685	0.175	0.796	0.736	0.755	0.510
HMMDetector	0.795	0.488	0.619	0.696	0.653	0.306
ICCDetector	0.440	0.153	0.741	0.552	0.643	0.286
MaMaDroid	0.721	0.086	0.893	0.798	0.817	0.635

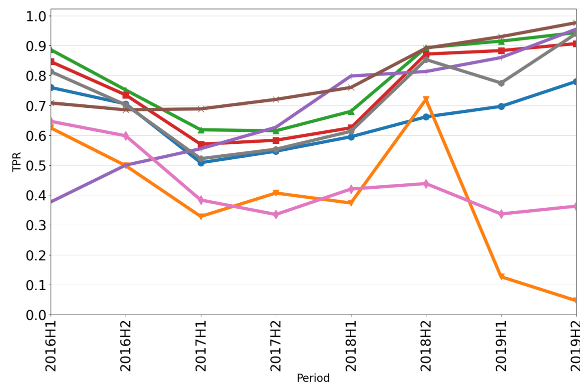
The overall performance of most of the methods under this scenario is characterized by an increment of both false positives and false negatives (see Table 4.14). Figure 4.3a provides a more detailed view of the evolution of the TPR, FPR and A_{mean} values throughout the period between 2016 and 2019. As can be seen, the performance of detectors changes in a notable manner from one period to another. Our hypothesis is that the popularity of some malware families decreases with time, and, at some point, they are replaced with newer families for which a detector may not have been trained. Also, at some point, an old behavior may become popular again, resulting in a sudden increase in the performance of that detector. This is why the lines in Figure 4.3c do not follow clear, decreasing trends. In addition, contrary to what is stated in [232], the incremental trend in the number of false alarms of detectors (see Figure 4.3b) indicates that goodware also changes its behavior over time. At any rate, our observations demonstrate the non-stationary nature of malware and goodware, and show how classic, batch-trained ML algorithms are not an appropriate solution for Android malware detection given the instability of their decisions.



(a) A_{mean} values



(b) FPR values



(c) TPR values

Fig. 4.3: Evolution of the performance of baseline detectors for the period 2016-2019

4.7 Discussion: Towards a Realistic Framework for Malware Detection

Based on the factors analyzed in the previous sections, this section presents a collection of ideas or recommendations to consider when designing and evaluating a realistic proposal for Android malware detection.

Undoubtedly, one of the main problems encountered when researching malware detectors for Android is the lack of reproducibility of many proposals in the literature. The datasets and code used in the experimental processes are rarely public, and the details provided in the papers are often not enough for a correct re-implementation of the methods. In addition to this general aspect, the experiments conducted in the previous section have highlighted the importance of using adequate datasets when training models and assessing their performance. Indeed, as we have seen, depending on the datasets used, the same model can show near-perfect performances or be almost irrelevant. Thus, we can conclude that the datasets and experimental scenarios considered in the literature are unrealistic and should be revised.

Related to the previous statements, our analysis has shown that the presence of highly similar apps in the datasets influences the performance of models. This also opens the possibility to perform evasion attacks against classifiers, for example, by developing malware apps with snippets of code from minor malware variants. During training, models tend to focus on large groups of duplicates, instead of trying to generalize the variety of apps in the data. During model assessment, duplicates are the cause of misleading performance indicators that tend to overestimate the detection ability of models. We can conclude that training and evaluation efforts should leverage on datasets without duplicates to improve and demonstrate the generalization capacity of models [128]. Training should be carried out after balancing the representatives within each of the classes, i.e., between groups of similar samples; to avoid biases and exploit the generalization capacity of ML models. Also, contextual evaluations that take into account the prevalence of different malware families or groups are desirable to describe the actual reasons behind the performance of detectors. The removal of duplicates has an additional advantage, as the implicit reduction in the dimensionality of the data is useful to speed up the labeling, training and evaluation processes.

In relation to the labeling of the apps, we have seen that VirusTotal is widely used, but relying only on the VTD reported by this tool entails some risks [335]. To begin with, there is a lack of consensus on the labeling of the apps and on the inclusion of greyware in the training and evaluation of models. We should bear in mind that the line between malware and goodware is not clearly defined. As we demonstrated, the choice of the labeling criteria highly influences the generated model and the detection performance, simplifying the problem as the VTD is increased. This lack of consensus also results in the omission of a large amount of greyware which lies in between. However, such apps are present in the Android app ecosystem [111, 22], so, realistic proposals should not ignore these type of apps but include them in their training and evaluation processes.

Also regarding labeling, a more general problem is that not all antivirus engines in VirusTotal are equally reliable, with some of them being correlated or specialized in specific types of malware [205]. In addition, two engines from the same vendor but specific for different platforms may differ [256]. Although reliable labels can be derived from the VTD [335], setting simple thresholds to the VTD assumes that all antivirus engines are equally reliable in all situations and makes no distinctions between them. In order to overcome these aspects, techniques such as crowd learning [303], which measure the relevance of the different antivirus engines present in VirusTotal, have been used. As a last problem with labeling, we have seen in the experimentation that the VTD, and thus, the labeling, changes over time [335]. Weakly labeled data and changes in the labeling can hinder the generation of robust classifiers, leading to detection errors.

Another important aspect is that malware and goodware apps are found in different proportions in the wild, depending on the source of the apps, but malware being the minority class [232]. Ideally, a detector should perform well when analyzing apps regardless of the source of the samples. Thus, proposals should be trained and tested assuming unbalanced datasets. Additionally, the adoption of suitable performance metrics for unbalanced scenarios, such as the kappa and A-mean metrics, should be contemplated as opposed to using others, such as the accuracy, that do not reflect the real performance of detectors in these contexts. We have demonstrated that classical ML supervised classification algorithms used for malware detection do not properly manage imbalance, as they expect data to be equally balanced in order to build a robust model [129].

Next, detectors have to work in a hostile scenario where attackers will try to evade them to infect a system. These are one of the first lines of defence, so security of malware detectors is important. In this Chapter, our evaluation was limited to simple evasion attacks based on the use of obfuscation techniques. We evidenced that most detectors are not designed with security in mind and are, in fact, vulnerable to these attacks. Throughout Chapter 5, we investigate the informativeness of the most common static analysis features when obfuscation is used, and propose a detector that is robust against obfuscation-based evasion attempts.

Finally, the malware detection problem is non-stationary, i.e., classes evolve over time and rarely show constant characteristics. A realistic detector should be able to cope with the non-stationary nature of Android apps. We demonstrated that the analyzed detectors, which are based on classical ML classifiers, are not able to manage such changes. We analyze and propose efficient techniques to tackle this aspect in Chapter 6 of this dissertation.

4.8 Conclusions

One of the main conclusions of this chapter is that authors of ML-based Android malware detectors tend to be very optimistic when designing and evaluating their systems, ignoring factors such as the presence of duplicates in the datasets, the lack of robust labeling methods, the presence of greyware, the imbalance between

malware and goodware, the existence of apps trying to evade detection and the evolution of apps. Our evaluation work has shown how these factors substantially affect the performance that can be achieved with the tested detectors. We have seen, therefore, that malware detectors are not ready for deployment in real environments. Another important problem we have studied is the lack of a common design and evaluation framework, stemming, among other things, from the unavailability of codes and standardized and appropriate datasets. This fact greatly complicates the reproducibility of experiments. Our contribution in this regard has been to release our data and codes so that they can be used by other researchers.

5

Dealing with obfuscation

As we described in Chapter 4, malware authors often employ obfuscation as a means to evade malware detectors that rely on static analysis features. However, we also showed how this crucial aspect has received insufficient attention within the domain of malware detection research. This lack of comprehensive assessments has led to a notable disparity in the literature, with some authors stating that detectors based on static analysis information are ineffective against obfuscation, while others claim their proposals to be obfuscation-resistant.

In this context, the primary aim of this chapter is to determine the extent to which the use of specific obfuscation strategies or tools poses a risk to the effectiveness of different Android malware detection approaches that rely on features extracted through static analysis. Our experimental results indicate that obfuscation techniques indeed impact static analysis features to varying degrees across different tools. Nevertheless, certain features maintain their validity for ML-based malware detection even in the presence of obfuscation. These findings present an opportunity for the development of obfuscation-resilient detectors utilizing static analysis features and, in consequence, we propose a robust Android malware detector that outperforms state-of-the-art solutions, even when facing obfuscated apps.

5.1 Introduction

The most typical approach of evasion techniques is to complicate the extraction of the information that explains the behavior of apps and that is used for detection [87]. In this regard, obfuscation is a security through obscurity technique that aims to prevent automatic or manual code analysis. It involves the transformation of the code of apps, making it more difficult to understand but without altering its functionality [269]. This characteristic has made obfuscation a double edged sword, used by both, goodware and malware authors. Developers of legitimate software leverage obfuscation to protect their code from being statically analyzed by third parties, e.g., trying to avoid app repackaging or intellectual property abuses [67]. Malware authors have seen obfuscation as a mean to conceal the purpose of their

code [80], preventing static analyses from obtaining meaningful information about the behavior of apps.

It may seem common sense that the application of any, or the combination of several, obfuscation techniques will make malware analysis relying on features extracted using static analysis fruitless. However, it is unclear to what extent this aspect is true. Some studies on Windows and Android executables have demonstrated that obfuscation harms detectors that rely on static analysis features. For example, packing[1] prevents obtaining informative features [4, 253], which are essential to train classifiers. Similar conclusions have been drawn for other forms of transformation [124, 209], showing a major weakness in Android malware detectors. However, other studies contradict what has been stated in the aforementioned works, proposing feature extraction techniques via static analysis that enable a successful identification of malware even when apps are obfuscated [279, 29, 103].

All of these works appear promising in demonstrating either the flaws or the strengths of static analysis features for malware detection. However, these discrepancies complicate the extraction of sound conclusions regarding the validity of static analysis features for Android malware detection. In addition, many of these works focus solely on the labels predicted by the detectors, without analyzing the effect of the obfuscation on the apps and/or features used to train them [247, 201, 124, 29, 209]. This additional feature-centered information is important to understand and explain why the detectors are working or failing when obfuscation is present, and is crucial for building more robust detectors. Finally, another evident flaw of some of these studies is the lack of details concerning their datasets and the configuration of their experimental setups [279, 103, 180, 308]. Apart from the lack of reproducibility, biases in the datasets may lean the results towards non-generalizable results. Therefore, the conclusions drawn from all these works may have limited applicability beyond the evaluated scenarios, and can be the cause of the contradictions found in the literature.

To the best of our knowledge, this work presents the first comprehensive study about the impact of common obfuscation techniques in the information that is obtained through static analysis to perform malware detection in Android with ML algorithms. The contributions of this chapter can be summarized in the following highlights:

- We provide an agnostic[2] evaluation of the strength, validity and detection potential of a complete set of features obtained by means of static analysis of APKs when obfuscation is used.
- We analyze the impact of a variety of obfuscation strategies and tools on static analysis features, providing insights about the use of these features for malware detection in obfuscated scenarios.

[1] Packing is a particular form of obfuscation which hides the real code through one or more layers of compression/encryption. At runtime, the unpacking routine restores the original code in memory to be then executed.

[2] In this context, we refer agnostic as an analysis carried out without focusing on a specific malware detection proposal.

- We propose a high-performing ML-based Android malware detector leveraging a set of robust static analysis features. We demonstrate the ability of this detector to identify goodware and malware despite obfuscation, outperforming the state-of-the-art.
- We present a novel dataset with more than 95K obfuscated Android apps, allowing researchers to test the robustness of their malware detection proposals.
- In spirit of open science and to allow reproducibility, we make the code publicly available at `https://gitlab.com/serralba/robustml_maldet`.

The rest of this chapter is organized as follows. Section 6.2 introduces the literature that has previously tackled obfuscation as a problem in malware analysis. Section 5.3 describes the construction of the app dataset and presents the features that are considered in our experiments. Section 6.7 evaluates the impact of different obfuscation strategies and tools in static analysis features, as well as their validity for malware detection. Section 5.5 is devoted to assess the robustness of our ML malware detection proposal. Section 5.6 includes a discussion of the main findings made along this chapter. Finally, we conclude this chapter in Section 6.9.

5.2 Related Work

The related work can be divided into two groups: (1) studies that analyze the vulnerabilities of malware detectors when obfuscation is present, and (2) works that propose novel malware detectors which are purposely robust to obfuscation.

5.2.1 Study of the Vulnerabilities of Malware Detectors

The works that evaluate the negative effects of obfuscation on Android malware detectors have mainly been carried out for black box malware detectors, i.e., the system or model is analyzed and evaluated based solely on its input-output behavior, without direct access to or knowledge of its internal workings. The first work of this type [247] studied how obfuscation impacts the detection ability of 10 popular anti-virus programs available in the VirusTotal platform. The work demonstrated that these detectors are vulnerable and loose their reliability in the identification of obfuscated malware. Similarly, in [201], 13 Android anti-virus programs from VirusTotal are assessed using different obfuscation strategies to modify malware. The results showed a meek improvement in detection accuracy concerning the findings of previous works [247] and proved that companies responsible of developing these tools are trying to counteract obfuscation. A more comprehensive analysis for 60 anti-virus tools in VirusTotal has been presented in [124]. Again, the work demonstrated the vulnerabilities of most detectors when facing obfuscated malware. However, this analysis shows that the success on bypassing detection highly depends on the obfuscation tools and strategies considered.

In the mentioned studies, the detectors are commercial products with unknown characteristics. Some other works have focused on assessing the impact of obfuscation in published ML based detectors. In [29], an analysis of the effect of obfuscation

in two detectors, one relying on static and the other on dynamic analysis features, is presented. It is shown that the performance of the detector using dynamic analysis features is not altered by obfuscation, contrary to the detector that uses static analysis features. However, authors indicated that this effect can be easily mitigated by including obfuscated samples during the training phase of ML models. In [209], eight state-of-the-art Android malware detectors leveraging static analysis features and ML algorithms are assessed using obfuscated malware samples. The authors demonstrated that obfuscation is a major weakness of these popular solutions, since all of them suffered a drop in their performance. One of the most recent and comprehensive studies is carried out in [4]. This work analyzes the effect of packing in ML malware detectors relying on static analysis for Windows executables. The conclusions drawn from the extensive set of experiments indicate that ML malware detectors for Windows fail to identify the class of transformed samples due to the insufficient informative capacity of static analysis features.

All these works prove the added difficulty that obfuscation entails for malware detection. However, most of them fail to provide explanations behind accurate or erroneous detections. In this sense, they treat the detectors as black-box tools and do not analyze the effect of different obfuscation strategies and tools on the apps and, specifically, on the features that will be used for training the detectors. This makes it difficult to extract meaningful insights and provides no useful information to build more robust classifiers.

5.2.2 Obfuscation-Resilient Detectors

A second group of proposals focuses on the development of obfuscation-resilient detectors, specifically designed to operate effectively in the presence of obfuscated apps. Two of the most relevant works in this regard are DroidSieve [279] and RevealDroid [103]. The former categorizes static analysis features as obfuscation-sensitive and obfuscation-insensitive based on theoretical aspects. Feature frequency is studied for different datasets with obfuscated and unobfuscated malware samples to support the idea that most changing features provide better information. In consequence, they proposed a detector that relies on the features of both groups, and exhibits good performance in terms of malware detection and family identification. The latter work argues against static analysis features such as Permissions, Intents or Strings for robust malware detection. Contrary to the authors of DroidSieve, they suggest that obfuscation-sensitive features do not provide useful information to detect malware. Instead, the authors propose a new set of static analysis features based on a backward analysis of the calls to dynamic code loading and reflection APIs. In this way, the functions invoked at runtime are identified, nullifying the effect of obfuscation and making the proposed detector obfuscation-resilient.

Two allegedly obfuscation-resilient detectors leveraging deep learning algorithms are presented in [180] and [308]. The authors of these works suggest that the capacity of deep learning to embed and extract useful information from the features is enough to tackle obfuscation. The first work relies on strings extracted from the app code. Strings are then transformed into sequences of characters to obtain an embedded

representation of the app that is then used for classification. Despite the excellent results reported for malware detection, the ability of the detector to identify obfuscated apps is based on (unproven) statements that are not specifically tested. The latter proposal incorporates obfuscation-sensitive and insensitive features, including permissions, opcodes and meta-data from ApkID[3], a signature-based fingerprinting tool. Similarly to the previous proposal, the obfuscation-resiliency of this work cannot be confirmed based on the results, since the effect of the use of obfuscation in the detector is based on theoretical aspects not specifically covered by the experiments.

The experiments carried out in all these works present some flaws that, in our opinion, put in question their capability. For example, most of them do not describe, or vaguely analyze, the composition of their datasets in terms of the number of obfuscated malware or goodware samples, as well as the tools and strategies considered to obfuscate the samples. Some articles focus their analyses exclusively on obfuscated malware, either for the training or evaluation of the detectors, but what about obfuscated goodware? How do detectors behave in the presence of such apps? The use of different obfuscation tools or strategies for malware and for goodware introduces biases in ML algorithms, since the generated models may associate obfuscation, or the use of a particular obfuscation tool, to a specific class in the data [4]. Additionally, experiments performed with malware and goodware captured from different periods can cause biases in the detectors [23]. These aspects may justify the good published results and cause contradictions concerning other analyses carried out for ML-based detectors [29, 209]. Finally, we also found that most of them do not provide enough details to reproduce their systems and thus, lack of reproducibility.

5.3 Dataset

For our experiments, we rely on a dataset containing obfuscated and non-obfuscated apps. From this collection of apps and by means of static analysis, we obtain a set of feature vectors that constitute the object of this study. This section describes how the app dataset is built and the types of features derived from the apps.

5.3.1 App Dataset

We use the malware and goodware apps from the dataset presented in Section 4.5.1 of this dissertation to derive new datasets for these experiments. Firstly, apps are filtered out to derive a new dataset free of obfuscated samples, which we call "filtered". To do so, we leverage APKiD[4], ruling out the samples flagged as "suspicious" of including obfuscation.

In a second step, we generated obfuscated versions of the apps in the filtered dataset. To perform this process, we used the DroidChameleon [246], AAMO[237],

[3] `https://github.com/rednaga/APKiD`
[4] `https://github.com/rednaga/APKiD`

and ObfuscAPK[17] tools. We chose these tools because (1) they are open source, (2) they provide a wide range of obfuscation techniques, and (3) they have previously shown to effectively evade Android malware detectors. Specifically, for each obfuscation tool, we try to obfuscate every app in the filtered dataset using six obfuscation techniques: Renaming, Junk Code Insertion, Reflection, Call Indirection and Encryption. The configuration of the tools was left as default for all techniques. The results of this process are summarized in Table 5.1.

Note that some tools failed due to errors during the APK decompilation process. It is worth noticing that there were more failures in the case of malware apps than in goodware apps. ObfuscAPK was the tool with the best success rate, correctly obfuscating an average of 85% of the apps. On the contrary, we were unable to obtain obfuscated samples when trying to apply Encryption with AAMO, due to bugs introduced in the code by this tool that prevent the APK from being rebuilt. The attempts to use Renaming with DroidChameleon were also unsuccessful due to an error in the implementation of the tool. For other techniques, DroidChameleon and AAMO had average success rates of 55% and 28%, respectively. During this process, we realized that for some apps all the tool-technique combinations failed, and thus these apps were removed from the filtered dataset. We ended up with a "Clean" dataset that consists of 4 749 goodware and 4 067 malware (presumably) non obfuscated samples.

Table 5.2 summarizes the different datasets that will be used in the experiments. The criteria for the composition of these datasets will be explained in Section 6.7.

- NonObf: It includes the non obfuscated versions of the apps for which we could not obtain an obfuscated version with all the tools for at least one technique, i.e., apps that can be obfuscated using a specific tool and technique but not with the remaining tools using the same technique.
- CleanSuccObf: includes the subset of non obfuscated apps present in Clean, but not in NonObf. That is, all the apps for which all the tools have worked for at least one technique.
- The remainder datasets (Renaming, JCI, CallIndirection, Reflection, and Encryiption) contain the obfuscated versions of the apps in CleanSuccObf for a particular technique using all the tools.

5.3.2 Feature Dataset

An app dataset has to be transformed into a dataset of feature vectors prior to perform malware detection using ML. Following a detailed literature analysis, we identified seven families of static analysis features that have proven to be useful for ML-based malware detection [296]. We used two well-known and widely used static analysis frameworks for Android to extract these features: Androguard [75] and FlowDroid [26]. Sources of these features include: the *classes.dex* and *AndroidManifest.xml* files, as well as the contents of the *res* and *assets* directories of APKs.

Table 5.1: Success rate of different technique-tool obfuscation combinations for the apps in the Clean dataset. The first part of the name refers to the tool used to obfuscate the apps, with *DC* for DroidChamaleon, *AA* for AAMO, and *OA* for ObfuscAPK. The characters after the underscore refer to the strategy followed to obfuscate the apps: renaming (*rnm*), junk code insertion (*jcins*), call indirection (*ci*), reflection (*refl*) and encryption (*encr*).

tool-technique	**#Goodware samples**	**#Malware samples**	**Obf. Success Rate**
DC_rnm	-	-	0%
AA_rnm	2244	1953	34%
OA_rnm	5690	4317	81%
DC_jcins	1855	1123	24%
AA_jcins	2289	2019	35%
OA_jcins	6003	4755	87%
DC_ci	3664	2209	47%
AA_ci	1337	1362	22%
OA_ci	6050	4765	87%
DC_refl	6200	3993	82%
AA_refl	1332	1402	22%
OA_refl	6080	4802	88%
DC_encr	5008	3746	70%
AA_encr	-	-	0%
OA_encr	6074	4814	88%

Table 5.2: Composition of datasets used in this work. The columns indicate the number of samples that comprise each set. The CleanSuccObf dataset contains the clean (original) apps for which we obtained obfuscated versions with all tools for at least one technique.

Dataset	**#Goodware samples**	**#Malware samples**
Clean	4749	4067
NonObf	1345	1211
CleanSuccObf	3404	2856
Renaming	3238	2868
JCI	1515	1008
CallIndirection	2118	1737
Reflection	2667	2484
Encryption	4790	4060

5.3.2.1 Permissions

Permissions have commonly been used as a source of information for malware detection in Android [24, 295, 91, 334]. In this category, we consider as features the full set of permissions provided by Google in the Android documentation[5], as well as the set of custom[6] permissions that developers may declare to enforce some functionality in their apps. Following this procedure, we extracted a set of binary features, each corresponding to the presence or absence of a given permission.

5.3.2.2 Components

An app consists of different software components that must be declared in the *AndroidManifest.xml* file. These elements have been widely used as a source of information for malware detectors [307, 24, 310, 91]. We extract a list of hardware and software components that can be declared using the *uses-feature* tag from the Android documentation[7], as well as every identifier for Activity, Service, ContentProvider, BroadcastReceivers and Intent Filters. In total, we obtained a set of 85 476 binary features, whose value is set to True or False for an app according to the presence of the feature in its *AndroidManifest.xml* file. We additionally derive seven frequency features accounting for the number of elements of each type in the app.

5.3.2.3 API functions

API libraries allow developers to easily incorporate additional functionality and features into their apps, being the main mean of communication between the programming layer and the underlying hardware. As such, analyzing the calls to methods of these libraries (API functions) constitutes a good instrument to characterize the functionality of apps, and, therefore, for malware detection. Following similar approaches to those proposed in the literature [307, 24, 334, 169], we extract a binary feature for each API method, and set its value to True if the app contains any call to that method within its code. In total, this set consist of 66 118 binary features.

5.3.2.4 Opcodes

The compiled Android code (Dalvik) consists of a sequence of opcodes. Opcode-based features provide insights about the code habits of developers as they represent fine-grained information about the functionality of apps [164]. Subsequences of opcodes, or simply n-grams, have been used for Android malware detection in

[5] `https://developer.android.com/reference/android/Manifest.permission`
[6] `https://developer.android.com/guide/topics/permissions/defining`
[7] `https://developer.android.com/guide/topics/manifest/uses-feature-element.html`

[147, 56, 153, 204]. Concerning the size of the subsequences, Jerome et al. [147] and Canfora et al. [56] observed that $n = 2$ offers a good trade-off between the size of the feature vector generated and the performance obtained by detectors. Therefore, we extract unique opcode subsequences of length 2 (or bi-grams) from the code of the apps, and create a feature to represent the number of appearances of each bigram in the code. The resulting vector contains a total of 25 354 frequency features.

5.3.2.5 Strings

The APK file strings are a valuable source of information for malware detection. In this regard, the most common strings include IP addresses, host names and URLs [24, 297]; command names [318, 329] and numbers [297]. We processed app files and found 2 425 892 unique strings. Following the procedure in [4], we observed that 98.5% of the strings were present in less than 1% of the samples. After removing these rare strings, we obtained 39 793 binary features, each representing the presence or absence of a specific string within the app files.

5.3.2.6 File related features

This type of features includes the size of code files and different file types inside the APK [116, 318, 297, 279]. We base our file type extractor on both, the extension of the file and the identification of the first bytes of the content (i.e., magic numbers) of files. The result is a new frequency feature for every unique combination of the extension (*ext*) and magic type (*mtype*), identified as *ext_mtype*. For files without extension, we use the complete file name instead. In total, this set consist of 65 986 frequency features per app.

5.3.2.7 Ad-hoc Features

As explained earlier, some specific detectors claim to use obfuscation-resistant features. We call the features used by these detectors that do not fall into any of the above categories ad-hoc features. They include: semantic features based on sink and source relationships in the code [329]; certificate information [279]; flags about the use of cryptographic, reflective, and command execution classes [116, 297, 169]; and resolved function names for native and reflective calls [103]. Due to the computational cost of obtaining these features, we limited the time spent computing them to 15 minutes per sample. The result is a set of 35 387 frequency features, each representing the number of occurrences of the feature within the app.

5.4 Feature Validity

As a first step in this study, we have designed a set of experiments to determine the robustness and detection ability when obfuscation is present of the seven feature

families described in the previous section. The first experiment analyzes the impact that different obfuscation strategies and tools have on the features. In the second experiment we evaluate the performance and stability of ML algorithms when using these features for malware detection.

5.4.1 Feature persistence

In this experiment, we aim to examine the impact of obfuscation on the features presented above. We analyze when and how much the features change in the presence of obfuscation. We highlight the disparities among obfuscation tools and how different implementation strategies to achieve the same obfuscation objective can affect the features.

To analyze these aspects, we calculate the feature *persistence* for each tool-technique obfuscation combination. This is done by determining the average level of overlap between the features of an original (clean) app and its obfuscated counterparts. To compute the feature overlap, we compare each pair of feature vectors calculated for an original app and its obfuscated version, and quantify the proportion of features with exact value matches. Note that for binary-featured representations (Permissions, Components, Strings and API functions), this is equivalent to computing the Jaccard index that measures the ratio between the shared elements and the total number of elements in the union of both feature vectors. Note also that, for frequency vectors, an increment or decrease in one unit or ten units has the same effect in this metric.

Table 5.3: Persistence of static analysis features when comparing clean and obfuscated apps using ObfuscAPK (OA), DroidChameleon (DC) and AAMO (AA).

	Renaming			**Junk Code Insertion**			**Call Indirection**			**Reflection**			**Encryption**			**Avg.**
	OA	DC	AA	OA	DC	AA	OA	DC	AA	OA	DC	AA	OA	DC	AA	
Permissions	0.972	-	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.932	0.799	1.0	1.0	1.0	-	0.977
Components	0.219	-	0.999	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	-	0.939
API functions	0.480	-	0.717	1.0	1.0	1.0	0.493	0.718	0.489	0.985	0.407	0.487	0.994	0.999	-	0.751
Opcodes	0.985	-	0.993	0.269	0.107	0.116	0.832	0.832	0.970	0.979	0.919	0.826	0.959	0.982	-	0.751
Strings	0.995	-	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.906	1.0	0.897	0.028	0.009	-	0.833
File Related	0.895	-	0.993	0.803	0.880	0.987	0.791	0.874	0.983	0.845	0.904	0.987	0.926	0.923	-	0.907
Ad-hoc	0.923	-	0.942	0.655	0.345	0.560	0.455	0.667	0.409	0.888	0.859	0.850	0.92	0.922	-	0.722

The results of this experiment are shown in Table 5.3. We find various degrees of persistence, in most cases over 0.8, with many exact matches between the feature vectors of clean and obfuscated APKs. Components and Permission features suffer the smallest changes when applying strategies such as Junk Code Insertion, Call Indirection, Reflection and Encryption (independently of the tool). Despite being affected by all techniques, File-Related features are also among the least affected on average. On the contrary, Ad-hoc, API functions and Opcode feature vectors change the most when obfuscation is applied. Nonetheless, the average persistence values for these features indicate that most fields (about 75%) are not affected by

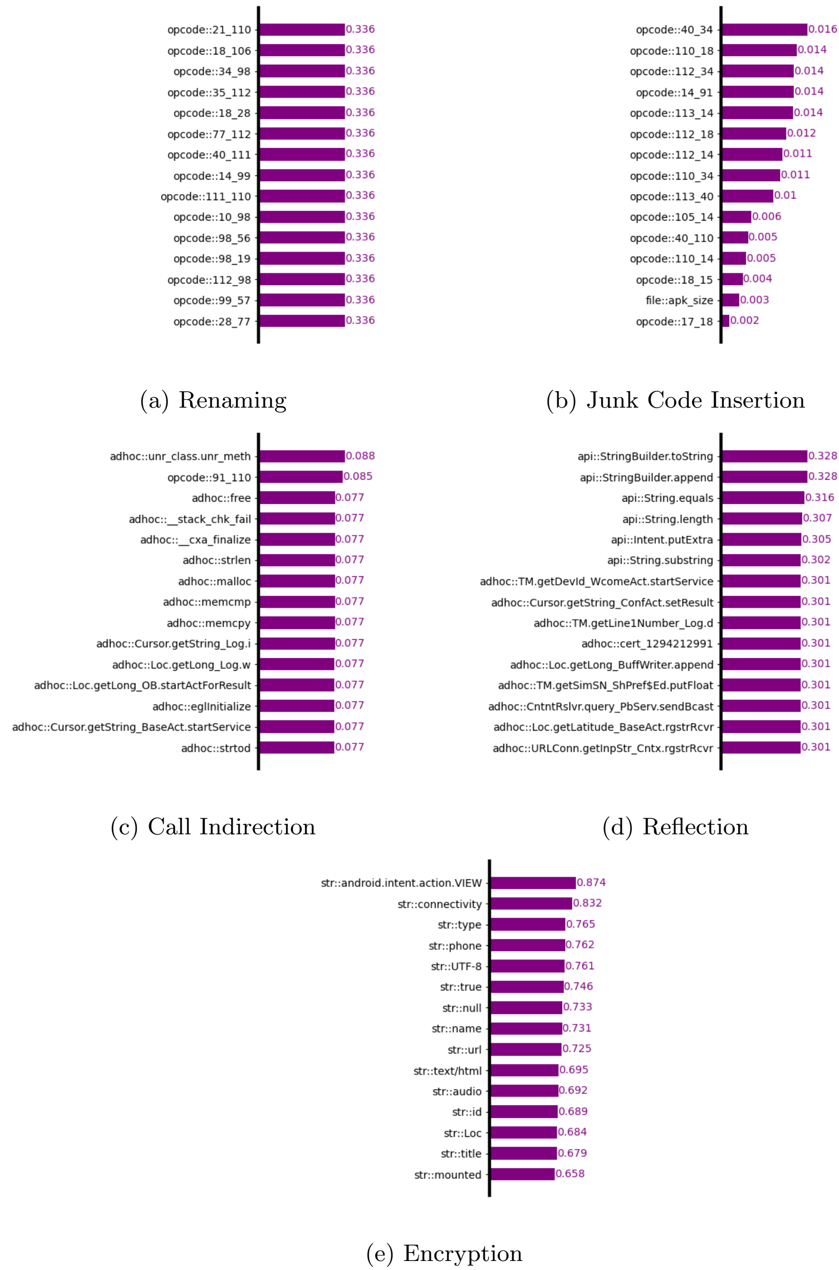
obfuscation. Therefore, in most cases, we conclude that the use of obfuscation is not reflected as a radical change in the feature vectors.

Persistence values refer to the proportion of features that remain unchanged, but do not tell us which particular features change the most when a tool-technique combination is applied. To shed some light on this regard, we selected the 15 features that change the most when obfuscation is applied. They may belong to different families. To obtain them, we measured the degree of discrepancy in the number of occurrences of each of these features, comparing the original application and the obfuscated version. To simplify the visualization, we show the results for each technique, averaging the discrepancy values for the three tools. The resulting rankings are shown in Figure 5.1. The name of each bar is the feature name (which includes its family). The number at the right of each bar is the degree of discrepancy, i.e., the average difference in the frequency of the feature between original and obfuscated versions of apps. Note that for easier interpretation, the scales are specific to each figure.

Regarding the persistence of the different feature families, Renaming mainly affected Components and API functions features, due to changes in the names of user-defined packages, classes, methods and fields. It also alters the declaration of custom permissions present in the code, since they depend on the name of the class where they are declared. However, as can be seen in Figure 5.1a, none of these features are among the 15 most affected, mainly because the names assigned to the classes are app-specific. In contrast, Opcode features are among those most significantly affected, due to changes in the order of methods when processing class files. This mainly changes the frequency of sequences that present invocation instructions (opcodes 110, 111 and 112).

In concordance with persistence values in Table 5.3, Figures 5.1c and 5.1b show that Call Indirection and Junk Code Insertion techniques are particularly detrimental for features based on code information, with Opcode and Ad-hoc features being the most sensitive to both types of obfuscation. In particular, Ad-hoc features are the most affected by Call Indirection (see Figure 5.1c) due to the added complexity in the analyses required for their extraction. This is the case of sink and source relations between API functions such as *Cursor.getString* and *Log.i*. Also, due to the addition of indirect calls, this technique increases the frequency of some opcode sequences such as "90_110" formed by an *iput* (90) instruction followed by an *invoke* (110). This technique involves adding hundreds of auxiliary (indirect caller) methods per class, either in separate or in the API classes inside the API. However, these methods are randomly named, which limits their impact (their popularity will be low). As shown in Figure 5.1b, Junk Code Insertion greatly alters the frequency of most Opcode sequences due to the inclusion of useless instructions, mainly *goto* (40) and *invoke* (110, 112, 113). The introduction of useless code also greatly impacts on the size of the APK file (File-related feature *file:apk_size*).

Reflection changes the persistence of features extracted from code analysis. This effect is clearly perceptible in Figure 5.1d. With this technique, the code is modified to hide the originally called methods and use reflective calls instead. This mainly affects the API functions that are called more frequently in the code, including



(a) Renaming (b) Junk Code Insertion

(c) Call Indirection (d) Reflection

(e) Encryption

Fig. 5.1: Top 15 of most changed features for each obfuscation strategy. The values on the right indicate the disparity or difference in the frequency of features between the obfuscated apps and their original versions. Results from the three tools have been averaged.

string-related functions such as *toString*, *append*, *equals*, or *length*. Ad-hoc functions are among the most changed due to the added complexity of identifying sink and source relationships that contain reflective code. Reflection also results in new string features that contain the class and function names invoked by reflection. However, these are declared once in the code, so their frequency is kept low. Permission features are affected because Reflection can hide the presence of protected API functions that require specific permissions to be granted.

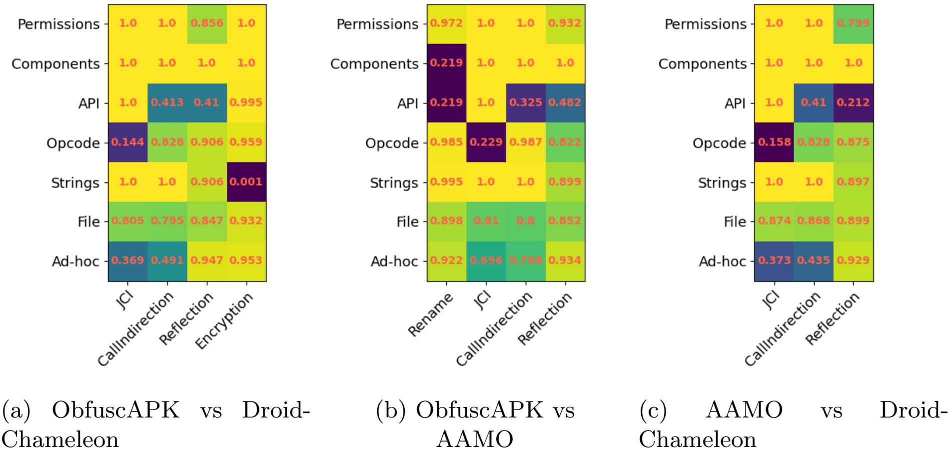
Encryption adds helper classes with the decryption routines that are used to hide user-defined strings and parameters. Therefore, API, Opcode, Ad-hoc, and File-related features are affected by the modifications introduced in the code. However, the main target of this technique are String features, as illustrated in Figure 5.1e and Table 5.3. These are heavily affected because their original values are encrypted. In this regard, the top 15 features most changed by encryption are strings related to the app's user interface (`UTF-8`, `phone`, `id`, `title`, `type`).

5.4.1.1 Differences between Obfuscation Tools

As seen in Table 5.3, changes in features depend on the tool used, mainly due to implementation particularities. Indeed, because of these peculiarities, obfuscation can even alter features that are not the primary target of the chosen obfuscation technique. Figure 5.2 depicts the average level of overlap between the features obtained for the same apps when obfuscated using different tools. Darker colors indicate less overlap in the obtained feature vectors, while lighter colors represent higher agreement. To better explain the differences obtained, we manually examined the code of these obfuscation tools as well as the features obtained from different samples. Due to space limitations and in order to make this chapter more readable, we omit very specific implementation details and limit our discussion to the more prominent differences.

We observed that of all the tools analyzed, none of them considered parameter randomization when implementing the different obfuscation techniques except for Junk Code Insertion. As such, for a given tool, the extracted feature vectors are only dependent on the input (app data). In other words, given the same input, a particular tool-technique combination will always return the same output. It is worth mentioning that all tools obfuscate (modify) the Android or Java libraries when this type of content is included in the APK, mainly due to poor checks during obfuscation. Since this code is not user-related, such changes may break the execution flow of apps.

The largest differences between obfuscation tools are present for API function and Ad-hoc features. This aspect is clearly perceptible in Figures 5.2. In the case of Reflection, we noticed that ObfuscAPK and AAMO perform a fine-grained checking when selecting the set of candidate function calls to be transformed, so that errors introduced by obfuscation are minimal. In contrast, DroidChameleon obfuscates calls whose package matches any of the prefixes included in a pre-defined list, without making any additional checks. In consequence, as shown in Figures 5.2a



(a) ObfuscAPK vs DroidChameleon (b) ObfuscAPK vs AAMO (c) AAMO vs DroidChameleon

Fig. 5.2: Feature overlap between every pair of obfuscation tools using different obfuscation strategies. Note that because Rename and Encryption do not work for DroidChameleon and AAMO, respectively, the corresponding columns are omitted.

and 5.2c feature overlap is low since DroidChameleon results in a higher number of transformed API calls with respect to ObfuscAPK and AAMO.

The way in which the files to be transformed are selected is also the explanation behind the differences observed between AAMO and ObfuscAPK with Renaming (see Figure 5.2b). By default, ObfuscAPK selects all the files within the APK as candidates for Renaming. This translates into changes in the content of files even if they belong to the Java library or the Android framework. Hence, features in the Components and API functions families are greatly modified by this tool. In contrast, AAMO performs some additional checks aiming to avoid modifying this type of content and presents a reduced impact on these features. Nonetheless, as shown by the persistence values for API-related features in Table 5.3, these checks are insufficient. For example, classes that are part of the "com.android" package are obfuscated because they do not match the name of the AAMO blacklisted "android" package.

The disparities observed for API function features with CallIndirection between the three tools (see Figures 5.2a, 5.2b and 5.2c) are due to the way intermediate methods are created. ObfuscAPK and AAMO insert the code of intermediate methods within the class file of the original calling method, whereas DroidChameleon adds this code to a (separate) helper class. As a result, methods inserted by ObfuscAPK and AAMO inside the class files of the Android framework are considered as API features by the feature extraction process. Since these tools use different naming conventions for these new methods, the resulting features do not overlap.

When applying encryption, ObfuscAPK and DroidChameleon use different algorithms and parameters. This explains the differences observed between both tools in Figure 5.2a. In particular, ObfuscAPK uses the AES cipher for encryption, whereas

DroidChameleon uses Caesar's algorithm. In both cases, the encryption key is hard-coded in their respective code.

5.4.2 ML Performance

We devise a second set of experiments to 1) analyze the ability of static features to detect malware, and 2) study the stability of these features for malware detection within ML algorithms in the presence of obfuscation. In all experiments, the RandomForests classification algorithm is used without any parameter optimization [50], as implemented in scikit-learn, a widely-used python library for ML [231].

The first scenario is focused on analyzing the predictive power of the different feature families in a fully non-obfuscated (clean) environment. For model training we use the NonObf dataset[8]. For evaluation purposes we used the apps from the CleanSuccObf dataset. Our objective is to evaluate the ability of an off-the-shelf classifier to approximate the class y of apps (malware or goodware) as a function of the original (non-obfuscated) features x_{orig} obtained from clean apps.

Table 5.4 shows the performance of the trained models. As can be seen, most features present high true positive rates (TPR above 0.8) and moderately low false positive rates (FPR below 0.2). Therefore, it can be concluded that most feature families provide enough information to enable effective malware detection using ML algorithms. This is particularly true for API functions and String features. On the contrary, the model generated using File-Related features performs similar to a random choice model (an A_{mean} value of 0.5) and therefore, we can say that these features are not suitable for the purpose at hand.

Table 5.4: Performance of static analysis features for malware detection using non-obfuscated apps for both training and evaluation. TPR stands for the True Positive Rate, i.e., the number of malware correctly identified. FPR stands for the False Positive Rate, i.e., the number of goodware erroneously identified as malware. The A_{mean} is the average of the TPR and the True Negative Ratio (1-FPR).

	TPR	**FPR**	$\mathbf{A}_{mean}$
Permissions	0.867	0.156	0.855
Components	0.808	0.157	0.825
API functions	0.928	0.081	0.923
Opcodes	0.884	0.252	0.816
Strings	0.907	0.082	0.912
File Related	0.265	0.197	0.534
Ad-hoc	0.768	0.143	0.812

[8] Note that an error during the obfuscation process of an app from this set for a given tool can be due to an error in the obfuscation tool, since the same app has been successfully obfuscated using other tools for the same and other strategies.

Table 5.5: Feature insensitivity, i.e., the overlap between the classifications made by the ML models for original and their obfuscated variants using ObfuscAPK (OA), DroidChameleon (DC) and AAMO (AA).

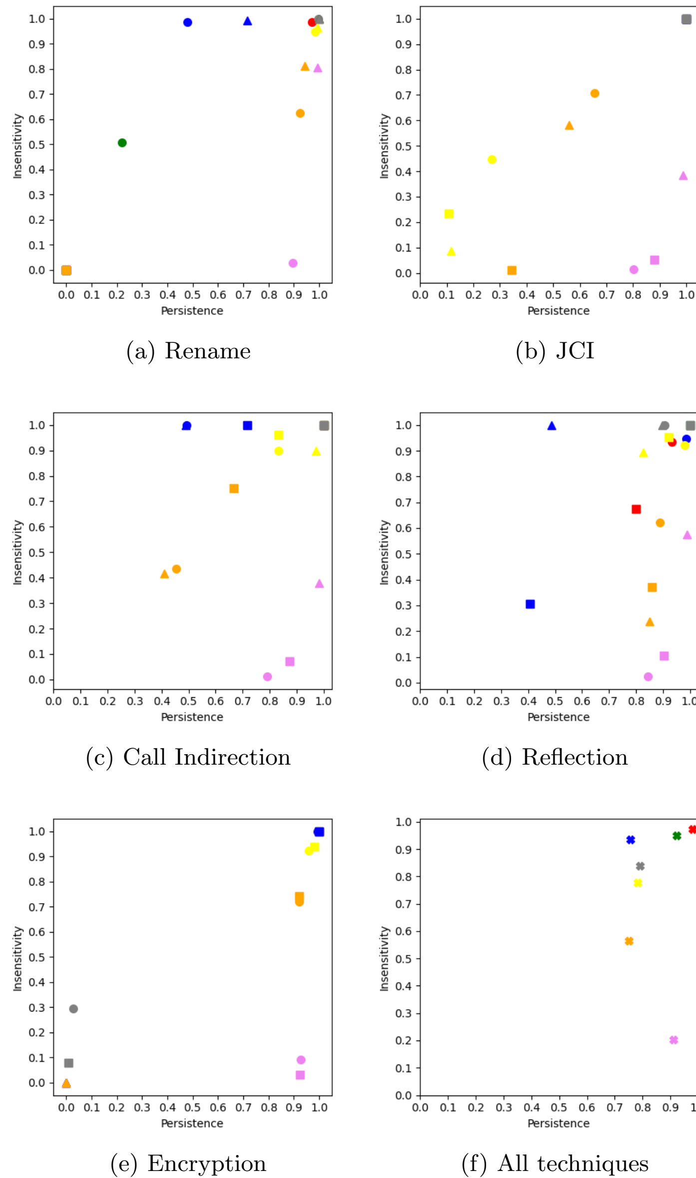
	Renaming			**Junk Code Insertion**			**Call Indirection**			**Reflection**			**Encryption**			**Avg. Over.**
	OA	DC	AA	OA	DC	AA	OA	DC	AA	OA	DC	AA	OA	DC	AA	
Permissions	0.986	-	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.934	0.674	1.0	1.0	1.0	-	0.968
Components	0.506	-	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	-	0.961
API functions	0.986	-	0.992	1.0	1.0	1.0	1.0	1.0	1.0	0.946	0.305	0.998	1.0	1.0	-	0.939
Opcodes	0.950	-	0.964	0.446	0.235	0.087	0.899	0.963	0.900	0.922	0.954	0.893	0.923	0.940	-	0.774
Strings	1.0	-	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.296	0.078	-	0.874
File Related	0.027	-	0.805	0.013	0.052	0.384	0.013	0.072	0.380	0.025	0.106	0.575	0.091	0.030	-	0.197
Ad-hoc	0.624	-	0.813	0.706	0.012	0.583	0.434	0.751	0.415	0.621	0.370	0.238	0.720	0.741	-	0.540

Even with high persistence values, small changes in feature vectors can lead to large changes in the performance of an ML algorithm. This may be the case if the small set of changed features is the most informative for a classifier and strongly influences its prediction. Consequently, this second scenario investigates the sensitivity of ML algorithms to the changes induced by feature vector obfuscation.

We use the ML models trained in the previous experiment (i.e., with clean apps from the NonObf set) and compile two separate evaluation sets for each obfuscation strategy. The first set, known as the obfuscated evaluation set, consists of (obfuscated) samples from the corresponding Renaming, JCI, CallIndirection, Reflection or Encryption datasets. The other set comprises the clean versions of those apps in the obfuscated dataset. By comparing the predictions made by the ML model for the clean and obfuscated versions of the same app, we can assess whether or not obfuscating an app can change the decision made by the ML model. We leverage the Jaccard index to compute the overlap between the predictions for the clean and obfuscated apps, and we refer to this measure as *insensitivity*. Thus, a high level of insensitivity indicates that the predictions made by a model are preserved even when obfuscation is applied to the apps.

The measured insensitivity values are compiled in Table 5.5. As can be seen, the decisions of ML models for most feature families are consistent. Permissions, Components or API functions result in stable predictions regardless of the obfuscation status of the apps, with insensitivity levels exceeding 90%. On the contrary, Ad-Hoc, Opcode and File-Related features exhibit high fluctuations in the decisions made by the models. This suggests a greater sensitivity of models to changes introduced by obfuscation in these features.

We wondered if the persistence of features when obfuscation is applied is somehow related to the insensitivity of ML models based on those features. In Figure 5.3, we represented the persistence and insensitivity values for the different feature families. As can be seen, in general, there is a high correlation between low persistence and high sensitivity, meaning that larger changes in the features vectors induce larger changes in the predictions of the ML algorithm. See for example Opcode features with JCI in Figure 5.3b and String features with Encryption in Figure 5.3e). However, high persistence values do not necessarily mean that the retained features



(a) Rename (b) JCI

(c) Call Indirection (d) Reflection

(e) Encryption (f) All techniques

Fig. 5.3: Relation between persistence and insensitivity to changes of the different features for each obfuscation technique and tool. Every color makes reference to a feature family, with red for Permissions, green for Components, blue for API functions, yellow for Opcodes, gray for Strings, violet for File-Related, and orange for Ad-hoc features; whereas symbols make reference to the values reported on each feature family for ObfuscAPK (circles), AAMO (triangles) and DroidChameleon (squares). The average of all tools is represented by the croos symbol.

are the ones that are more helpful to the ML models in making accurate predictions. For example, Ad-hoc features show high persistence values when applying Reflection (changed features are only 16% of the total). Still, the insensitivity is rather low, indicating those include the features that play an important role in the accuracy of predictions (see Figure 5.3d). Another example is File-related features, which show the most irregular behavior for ML models despite the small proportion of features altered by obfuscation (10% on average as shown in Figure 5.3f). In this regard, in the previous scenario we evidenced that File-related features lack informativeness for detection (see Table 5.4).

The previous results highlight an important finding: while persistent features are commonly considered reliable predictors for malware detection, persistency is not the sole factor influencing the robustness of the detection model. On the contrary, high insensitivity values implicate a high persistence on features, so it is a more adequate indicator of robustness. Therefore, it is crucial to carefully examine the impact of obfuscation-induced changes on the informativeness of the features, as even small changes can significantly impact prediction performance. In the next section, we explore the selection of different feature vectors based on ML performance and feature insensitivity to changes to develop robust malware detection models.

5.5 Robust Malware Detection

We hypothesize that it is possible to build a robust classifier (one with accurate predictions when dealing with both clean and obfuscated apps) by using features that are both relevant (generate good models with clean apps) and insensitive (the decision of the classifier does not change between the clean and the obfuscated version of an app). We call these *robust* features. On the contrary, features that obtained low insensitivity values (i.e., are highly sensitive) or are irrelevant for ML models are prone to cause fluctuations in the predictions of ML models when obfuscation is used.

To select a set of robust features based on the previous statements, we use the A_{mean} metric reported in Table 5.4 and the average feature insensitivity reported in Table 5.5. Specifically, we set three thresholds (0.8, 0.85, and 0.9) for both metrics (A_{mean} and feature insensitivity values) as criteria for selecting different sets of robust features – to be used for training and testing ML-based detectors. Table 5.6 shows the feature families selected at each threshold: the strictest threshold selects only the API functions family (A), the intermediate threshold selects API functions, Permissions, and Strings (PAS), and the lowest threshold selects Permissions, API functions, Components, and Strings feature families (PACS). Given the large number of features in the selected groups (particularly in PAS and PACS), we rank the features of each family based on the relevancy value computed by the corresponding RandomForest models from Section 5.4.2, and select only the best 2 000 features of each family. This selection does not apply to the Permissions family, as it only includes 683 features.

Three RandomForest classifiers are trained using the apps in the NonObf dataset (which does not include obfuscated samples): one for A features, another one for PAS features and the third one for PACS features. For evaluation, we use the CleanSuccObf dataset for the non-obfuscated scenario, whereas for the obfuscated scenario, we use the apps from the Renaming, JCI, CallIndirection, Reflection and Encryption datasets. The prediction performances of these three models are summarized in Table 5.7. As expected, results are good for the tests without obfuscation, with true positive rates over 90% and low ratios of false positives (under 8%). When tested with obfuscated apps, the model trained with API function features (A) showed a performance reduction of 3% in terms of A_{mean} with respect to the use of the non-obfuscated versions of the same apps, mainly due to the effect of Reflection. The use of additional features (PAS, PACS) seems to provide valuable information to models: with PAS features, we can observe a 2% reduction in the number of false positives, while the ability to correctly detect malware increases 3% with respect to the performance numbers obtained for the model using API functions. The addition of Components features (PACS) did not improve the performance with respect to PAS, obtaining 2% fewer true positives and a reduction of 1% in the number of obfuscated goodware being misclassified. Therefore, the best model is PAS, i.e., the one that uses Permissions, API functions and Strings.

For comparison, we also evaluate the performance of our RandomForest classifier with PAS features against RevealDroid [103], a robust state-of-the-art malware detector, and Drebin [24], the detector that showed the best performance throughout Chapters 4 and 6. Both detectors use their own sets of static analysis features and ML algorithms. RevealDroid features include API function and package counts, native calls extracted from binary executables and function names resolved from reflective and dynamic code loading calls. These account for a total of 59 072 features that are used to train a RandomForest model to perform malware detection. The features used by Drebin comprise declared and requested permissions, app components, hostnames, IPs, commands and suspicious and restricted API functions. This totals 253 881 binary features that are used to train a linear Support Vector Machine (SVM) goodware-malware classifier.

Table 5.8, shows the performance of our best model (RandomForest with PAS features) against RevealDroid and Drebin. As can be seen, PAS outperformed both state-of-the-art detectors for the non-obfuscated and obfuscated scenarios. With obfuscated apps, our robust proposal presented a 1% and a 6% higher detection rate, and 4% and 8% lower malware misclassification, with respect to Drebin and RevealDroid. These good numbers demonstrate that using a small set of Permissions, API functions and Strings is enough to perform malware detection in Android. In contrast to RevealDroid and Drebin, which mainly rely on Strings and API features, PAS considers a more balanced feature vector and hence, it is more robust against different obfuscation strategies. This experiment demonstrates that obfuscated malware and goodware can be identified using off-the-shelf ML algorithms and features obtained from static analysis, even without providing the algorithms with any information about the strategy or tool used to obfuscate apps.

Table 5.6: Features selected for robust malware detection based on different thresholds for the A_{mean} and feature insensitivity.

Threshold	Feature types	#Features
0.8	Permissions, API functions, Components, Strings	6 683
0.85	Permissions, API functions, Strings	4 683
0.9	API functions	2 000

Table 5.7: Performance of different robust feature combinations for ML malware detection. A, stands for the model using exclusively API functions. PAS, refers to proposal using Permissions, API functions and Strings, whereas PACS uses Permissions, API functions Components and Strings.

	Non-Obfuscated			**Obfuscated**		
	A	**PAS**	**PACS**	**A**	**PAS**	**PACS**
TPR	0.928	0.920	0.930	0.858	0.889	0.876
FPR	0.081	0.065	0.065	0.060	0.044	0.035
$\mathbf{A}_{mean}$	0.923	0.927	0.932	0.898	0.922	0.914

Table 5.8: Performance of robust ML detectors based on static analysis features. PAS refers to our robust detection proposal using Permissions, API functions and Strings.

	Non-Obfuscated			**Obfuscated**		
	RevealDroid	**Drebin**	**PAS**	**RevealDroid**	**Drebin**	**PAS**
TPR	0.856	0.914	0.920	0.832	0.876	0.889
FPR	0.117	0.103	0.065	0.12	0.088	0.044
$\mathbf{A}_{mean}$	0.869	0.905	0.927	0.856	0.893	0.922

5.6 Discussion

The experiments carried out in this chapter evidence that, as commonly assumed [317], static analysis features can be affected by specific obfuscation techniques. Feature persistence showed that all the feature families are affected by at least one obfuscation technique. Among them, the features obtained from the manifest of applications proved to be the most stable. Nonetheless, and contrary to what is commonly argued [32], our experiments also demonstrate that static analysis features can be a reliable source of information for ML malware detection. In this regard, we observed that some obfuscation strategies can result in additional features while leaving the original features unaltered. For example, this is the effect of CallIndirection in API functions, or Reflection in Strings. In most cases, the impact of obfuscation is limited to less than 20% of all the features derived from the samples (for example, at most 20% of the features are affected by Call Indirection and about 15% of them are altered by Reflection).

We also observed that the alterations caused by obfuscation on the features vary significantly between different tools, mainly due to implementation particularities.

However, the lack of randomization in these tools makes them produce the same output for the same input value, even for different source apps or executions of the same tool. Such behavior is useful to hide the explicit information provided by, for example a class name, but it is insufficient to conceal the intrinsic information, i.e., relationships between features, such as correlations. This means that apps that contain a similar characteristic, when obfuscated using the same tool, will maintain similar relations between the obfuscated values than for the original (unobfuscated) features. Additionally, most obfuscators need to improve the implementation of some obfuscation techniques due to the high failure rates they present. From the point of view of the users of these tools, these aspects pose a limitation of obfuscators.

The performance analysis of ML models trained with static analysis features revealed that some feature types (families) typically proposed for malware detection [226], such as file-related features, are not effective in differentiating malware. This experiment, conducted on non-obfuscated applications, identified API functions and Strings as the most informative features for malware detection, achieving detection rates of over 90% and a low false positive rates of 8%. We analyzed the impact of obfuscation-induced changes on the informativeness of features and showed that even small changes can have a significant impact on performance. Therefore, feature persistence should not be considered as the sole criterion for robust malware detection. This finding demystifies a common assumption in the Android malware detection field, which is to consider highly persistent features as robust.

By combining features that exhibited high insensitivity to changes and that presented high ML accuracy with non-obfuscated apps, we demonstrated that ML-based malware detection using static analysis features can be robust in the presence of obfuscation. Remarkably, this remains true even in scenarios where no knowledge about the obfuscation techniques applied to the apps is assumed, i.e., the obfuscated data is not taken into account during the training process. In this scenario, our proposed robust detection approach based on a stock classifier outperformed RevealDroid, the current state-of-the-art obfuscation-resilient detector, and Drebin, the best proposal for malware detection in Android according to a recent comparative [209]. Specifically, our detector achieved 92% of correct classifications, compared to 89% and 85% for Drebin and RevealDroid, respectively. This result shows that Android malware detection goes beyond the selection of a ML algorithm.

Finally, we are aware that the work done for this chapter has some limitations. The main one is that our analysis is limited to individual obfuscation strategies. However, these strategies can be combined to increase the probability of evading detectors. Also, the order in which these obfuscation strategies are combined affects the results obtained (note that the information hidden by a previous obfuscation technique also becomes invisible for the next obfuscation strategy). Evaluating all these combinations would drastically increase the number of scenarios to be evaluated. The cost of the required experiments would be impractical because: (1) samples would have to be obfuscated using combinations of different strategies and tools, and (2) feature extraction and model training would have to be performed on the resulting obfuscated samples. In addition, we believe that such an extensive

analysis would also make it difficult to draw clear conclusions about the impact of each individual obfuscation strategy. In this respect, this work can be seen as a first step in investigating the impact that the combination of different obfuscation strategies can have on static analysis features.

5.7 Conclusions

This chapter delved into the effectiveness of static analysis features for ML-based Android malware detection in the presence of obfuscation. To perform this assessment, we generated a variety of datasets by applying different obfuscation strategies to apps with the help of three state-of-the-art obfuscators. Seven families of static analysis features were defined and evaluated throughout an extensive set of experiments. We identified which families are more persistent when obfuscation is applied, and which families are more informative for correct Android malware detection. Based on these findings, we proposed the use of Permissions, API functions and Strings for ML-based malware detection. A stock implementation of the RandomForest classification algorithm using these robust features was used to generate a ML model able to separate malware from goodware with a remarkable success rate, without any prior knowledge of the specific obfuscation techniques applied to apps. In particular, it correctly identified 89% of evasion attempts with a low false positive rate of 4%, outperforming the current state-of-the-art solution for obfuscation-resistant Android malware detection.

6

Dealing with concept drift

In the Chapter 4, we evidenced that the rapidly evolving nature of Android apps poses a significant challenge to static batch machine learning algorithms employed in malware detection systems, as they quickly become obsolete. Nonetheless, the existing literature has paid limited attention to this issue, with many advanced Android malware detection approaches, such as Drebin, DroidDet and MaMaDroid, relying on static models. In this chapter, we will show how retraining techniques are able to maintain detector capabilities over time. Particularly, we will analyze the effect of two aspects in the efficiency and performance of the detectors: 1) the frequency with which the models are retrained, and 2) the data used for retraining. In a first set of experiments, we will compare periodic retraining with a more advanced concept drift detection method that triggers retraining only when necessary. In the second set of experiments, we will analyze sampling methods to reduce the amount of data used to retrain the models. Specifically, we consider fixed size windows of recent data and state-of-the-art active learning methods that select those apps that help keep the training dataset small but diverse. A third set of experiments shows how the combination of change detection and sample selection techniques results in very efficient retraining strategies which can be successfully used to cope with concept drift while maintaining the performance of state-of-the-art static Android malware detectors.

6.1 Introduction

Drift, which refers to the phenomenon where the statistical properties of the data being analyzed change over time, can be caused by data drift and/or concept drift. Data drift refers to changes which occur in the distribution of the input data over time, whereas concept drift or model drift is caused by changes in the relationship between the input data and the outcome of models, i.e., the conditional probability distribution of the class variable given the input [102]. Even if both drift types are interesting and deserve analysis, it has been demonstrated that concept drift is an urgent issue in Android malware detection since it causes the trained static ML

models to experience a steady decrease of their performance over time [232, 209, 63]. In this sense, in the rest of this chapter, whenever we mention the term drift, we will refer to concept drift.

It is evident that the Android app ecosystem has an evolving nature, because for example, new types of malware appear or new software features are added to the development framework [209]. However, most current anti-malware research solutions for Android rely on batch ML algorithms [190]. Under laboratory conditions, these algorithms have demonstrated extraordinary malware detection rates with low numbers of false positives, which make them a promising solution against malware [289]. However, batch ML algorithms are designed for static environments. They are used to train models offline on large datasets of labeled samples of malicious and benign apps, which are then used to enable accurate detection of new, previously unseen malware. Therefore, detectors that rely on these algorithms quickly become obsolete and lose effectiveness due to concept drift [102, 38].

In recent years, concept drift management methods have emerged as a promising solution to the challenges posed by drift in non-stationary apps [193] and in a variety of domains, including fault diagnosis [338], credit card fraud detection [48], network intrusion detection [207], and game recommender systems [7]. Concept drift management methods can be classified into two major groups: (1) retraining, which consists of replacing old models with new ones trained on the latest available data, and (2) incremental algorithms, which continuously update models as new data arrives. While incremental solutions are specific learning algorithms, retraining offers the advantage of being an agnostic approach that can be applied to any ML-based detector.

For Android malware detection, several researchers have proposed adaptive solutions to overcome the challenges posed by concept drift, either relying on incremental algorithms [215, 309] or retraining procedures [155, 117]. These algorithms propose completely novel detection approaches and ignore the relevance of most available state-of-the-art Android malware detectors, which rely on static analysis of code to extract the features that represent the apps, and leverage batch ML algorithms to perform detection [190]. At this point, it remains interesting whether these existing static detectors can be enhanced and adapted to changing scenarios using simple retraining mechanisms, avoiding the need to develop new detectors.

The successful implementation of retraining on existing detectors hinges upon a series of critical implementation decisions. These decisions involve establishing an retraining policy that determines when and with what data to perform the model retraining and replacement operations [298]. An inadequate retraining policy may result in unnecessary, too frequent, or insufficient retraining operations that render the model unable to adapt to changes in the distribution of data [31, 44]. Equally crucial is the selection of representative data reflecting current trends in the distribution but without forgetting reoccurring or stable patterns. New data has to be continuously stored, analyzed (sometimes manually) and labeled prior to being used for retraining Android malware detectors. Moreover, as the volume of the new incoming data increases, the storage, labeling efforts and computing requirements for retraining also increase proportionally [281].

The purpose of this chapter is to investigate the potential of retraining as a valid approach to enhance state-of-the-art batch Android malware detectors. Indeed we focus on retraining existing detectors and analyze techniques that reduce the cost of retraining. Particularly, we focus on two critical aspects: (1) the frequency of retraining and (2) the data used for this operation. Since the factors that cause drifts and thus, model aging, could be diverse and variable, model performance is monitored to trigger an update procedure whenever a degradation of the performance is observed. Regarding the training set used for model updates, we propose strategies to keep its size small and reduce the cost of labeling new data. Thus, minimizing the cost of retraining supervised models. Through a comprehensive set of experiments, we demonstrate that retraining offers a practical solution to address concept drift in solutions that use batch ML algorithms for Android malware detection.

The rest of this chapter is organized as follows. Section 6.2 analyzes the literature related to the present work. Section 6.3 introduces batch Android malware detection and how retraining can easily be applied to achieve model evolution. Then, in the next two sections, we focus on the specific methods that we analyze in this chapter to determine the retraining frequency (Section 6.4) and the data used for retraining (Section 6.5). Section 6.6 presents our experimental setup, introduces the three state-of-the-art batch Android malware detectors used in our experiments and describes the evaluation procedure followed for the analysis. Section 6.7 presents the obtained results and, finally, we discuss the main findings of our work, future research lines and conclude this chapter in Section 6.9.

6.2 Related Work

Learning in evolving environments requires defining two main aspects: 1) the mechanism used to update the model and 2) the data used to update the model. In this section, we briefly review the related proposals in the area of Android Malware detection, considering these two axes.

6.2.1 Adaptative Malware Detectors

As mentioned, the first decisive aspect when building a classifier in environments with drift is the mechanism used for adapting the model. Indeed, among the proposed adaptable Android malware detectors, we can find incremental learning algorithms that update their models with each data point, or retraining approaches, that train new models and replace the existing ones.

In a recent work [117] propose the use of a pool of batch RandomForest classifiers and an anomaly detection model fed with system call features. Detection is performed by majority voting the output of the models. Whenever models in the pool disagree, the anomaly detector is used to conclude the class of samples. In order to enable model adaptation, true labels are assumed to be known and the worst performing RandomForest model from the pool and the anomaly detector are retrained at fixed time chunks. In [215], the use of an incremental learning detector

that leverages contextual API call information as the feature set is proposed. The model is updated with every incoming sample. However, it assumes that the true label of every sample is known at real-time. The detector proposed in [155], uses a pool of Convolutional Neural Networks (CNN) fed with sequences of method, object and field names invoked in the code. Retraining is performed at fixed time chunks and using only samples for which the predictions are sufficiently reliable, so that labels obtained with majority voting of the pool are assumed to be accurate. In each retraining round the entire pool of CNN models is replaced. In DroidEvolver [309], a pool of incremental linear models is presented. For updates, models with low agreement decisions with respect to the rest of the models in the pool are adapted. For labeling the data, the approach uses pseudo labels obtained through majority voting of model decisions.

As mentioned in the introduction, all these approaches are completely novel detectors, which do not leverage any previously published state-of-the-art batch detector, at least not directly. In this sense, the difference between these works and our proposal is that we attempt to directly use the existing research using model agnostic retraining policies to enhance or maintain their performances when concept drift is present. Additionally, these proposals present issues related to the labeling of samples. For instance, using pseudo labels computed from model decisions has been shown to cause model contamination over time [152], while obtaining true labels incurs a cost that is often overlooked.

6.2.2 Out-of-Distribution Samples

The second aspect that must be taken into account when using retraining policies in drifting environments is the selection of data used for retraining. This data must be representative of the current concept, but the cost of labeling this data and retraining the model is proportional to the amount of data we use in this process. In this sense, some data selection strategies have been proposed in the Android literature.

The most common approach is to use the confidence of the current model in the prediction of a new sample as a way to analyze whether this new sample has been generated by the same probability distribution or not [313]. Confidence of a new sample can be measured by analyzing the consensus of several classifiers when predicting its class. In [309] and [330], low confident samples (for which models disagree the most) are used to update the models. Contrary to these approaches and despite it being potentially detrimental to the adaptation ability of models, in [215] and [155] low-confident data is treated as noisy and discarded from the update process to avoid model contamination when using pseudo labels. Similarly, [36] presents a decision rejection framework which aims to keep model decisions accurate over time by discarding unreliable model decisions for drifting samples. The framework presents a non-conformity measure which identifies drifting samples with respect to a set of reference samples used to train the model.

Other authors have proposed using specific models based on clustering ideas. [314] uses a neural network based on contrastive learning to group samples into either goodware or a specific malware family. A sample is identified as drifting if it lies far from all the identified groups in a certain retraining step. This proposal has been recently improved in [62] using a hierarchical contrastive learning classifier that ranks samples according to the fitness of the CL embedding and the prediction score of the classifier. The aim is to provide a more robust drifting sample selection in unbalanced scenarios.

All these OOD (out-of-distribution) selection proposals, focus on identifying the best samples to increase the detection ability of models. However, none of them can be directly used in a simple retraining framework that is model agnostic (i.e., is built over any detector). Additionally, they are general approaches that do not leverage the particular behavior of the Android environment to design specific sample selection strategies. In this chapter we will analyze, CL approaches and uncertainty sampling as model-agnostic retraining policies, and an ad-hoc sample selection method specifically designed for this problem.

6.3 Preliminary Concepts

We have discussed how most of the published literature on Android malware detection ignored concept drift as a foundational feature of Android malware detection. This section briefly describes how malware detection is typically performed using batch ML algorithms, as well as how these state-of-the-art detectors can be integrated into a retraining pipeline.

6.3.1 Batch Malware Detection

Typically, the Android malware detection process using batch ML consists of three main phases: a preprocessing stage, a training phase and a prediction phase [289]. This process is depicted in Figure 6.1. To begin with (preliminary step), a set of apps is required, and two tasks must be carried out. First, all the apps must be labeled. The labeling process consists of analyzing the code, metadata, and app behavior to identify any suspicious activity or known malware signatures, tagging the apps in the dataset as goodware or malware. Additionally, in this preprocessing stage, apps are examined, extracting the features indicative of their functionality and representing them in a structured manner. Examples of these features include permissions, function names, strings in the code, etc. Once the app labels and their features are obtained, in the training phase, ML algorithms help determine the most characteristic patterns of goodware and malware. As a result of this training stage, a ML model capable of predicting the class label (goodware or malware) of new apps is obtained. Finally, the prediction phase consists of extracting the features identified during the training phase from a new incoming app. Afterwards, these features are fed into the previously trained ML model so that it determines whether the app is goodware or malware.

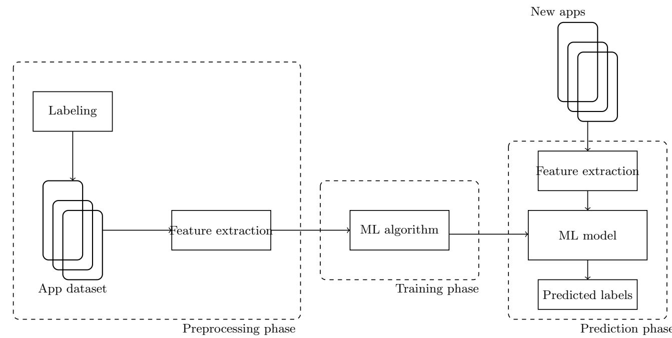


Fig. 6.1: Diagram of the batch learning process. (1) Preprocessing phase: a structured feature set and a label is obtained for each app; (2) Training phase: the structured and labeled training dataset is used to generate a model using ML algorithms; (3) Prediction phase: the generated model is used during the prediction phase to determine the class of new apps.

6.3.2 Retraining for Batch Malware Detectors

Retraining mechanisms consider a detector as a black box tool. This means that any existing batch detector can be integrated into the retraining process without modification. Figure 6.2 depicts how the retraining mechanism can be integrated into any existing detector. In order for new models to correctly represent the current data distribution, the training data has to be continuously updated with representative apps. Since Android malware detectors rely on supervised algorithms, these apps must be labeled. Retraining is signaled by a supervisor. Whenever the signal is raised, a new model is trained to replace the old one. This involves preprocessing all (or some) apps in the dataset to extract their features, training the new model with this information, and replacing the old model.

A very simple retraining policy is to activate the update process at fixed time intervals, for example, once a month. It can also be triggered whenever a certain number of new labeled apps become available, e.g., when 10 000 new apps have been identified. Nonetheless, the most effective strategy would be to trigger retraining whenever a drift is detected. The supervisor can monitor the performance of the model, or measure the degree of dissimilarity between training apps and incoming apps. In the following sections, we investigate the impact of some of these retraining strategies, as well as the impact of different retraining data management policies on the efficiency of batch Android malware detectors. Particularly, we focus on two mechanisms: fixed-period retraining and using a monitor that identifies changes to trigger updates. In addition, we explore three approaches for choosing the data used for retraining the models: a forgetting mechanism that discards old apps, three active learning methods that select highly-relevant data and a problem-specific sample selection technique based on clustering.

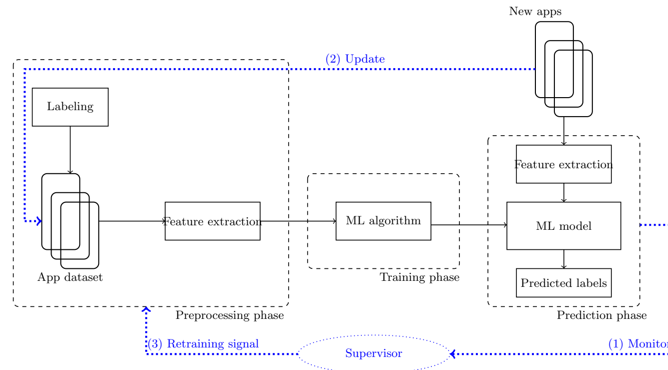


Fig. 6.2: Diagram of the batch learning process with retraining. The supervisor firstly monitors when changes take place, once a change is detected, the data is updated to reflect the current trend and a model retraining signal is raised. This trains a new model with the updated data that is used to replace the old model.

6.4 Retraining Frequency

In this section, we discuss the two different retraining policies mentioned above: (1) scheduling the update operation periodically and (2) using a change monitor that triggers the update when the performance of the detector drops.

6.4.1 Fixed Period Retraining

A naive update policy is to retrain Android malware detectors in batches using a fixed periodicity: weekly, monthly or any other. This method has several advantages, including ease of implementation and predictability. By following a fixed schedule, the system can regularly retrain the model to keep it up-to-date with the latest malware trends and behavioral patterns. However, this approach also has some limitations, because the rate of change of the data distribution might not be uniform or periodical. Due to the unpredictability of changes in Android data, choosing a fixed update frequency may be suboptimal. If the time between updates is long, the model may miss malware that has appeared and lasted for a short period of time. On the other hand, a high retraining frequency would eliminate this problem, but would result in unnecessary costs if changes in the data distribution are slow [98].

6.4.2 Change Detection Mechanisms

An alternative update policy to fixed period retraining is to use a change detection mechanism which monitors the current data or the performance of the model, triggering an update round only when there is evidence of change.

For this purpose, in this chapter we consider the Page-Hinkley (PH) test [224, 132], a popular (and easy to implement) drift detection algorithm that detects

changes by monitoring the performance of the model. The PH test has several advantages over other change detection methods. First, it is non-parametric and does not make any assumptions about the underlying data distribution. Secondly, it is computationally efficient and requires minimal memory, which makes it suitable for monitoring high-speed data streams. Finally, it is also robust to outliers and can detect gradual changes in the data distribution [44].

$$C_n = \begin{cases} 0 & \text{if } n = 1 \\ \min\left(0, C_{n-1} + (A_{mean_n} - \bar{x}_{n-1})\right) & \text{if } n > 1 \end{cases} \tag{6.1}$$

$$\bar{x}_{n-1} = \frac{\sum_{t=1}^{n-1} A_{mean_t}}{n-1} \tag{6.2}$$

$$PH_n = \begin{cases} 1 & \text{if } \lambda + C_n < 0 \\ 0 & \text{if } \lambda + C_n >= 0 \end{cases} \tag{6.3}$$

The PH test is applied as follows: it periodically (or whenever a certain batch of new instances are obtained) monitors a test value calculated based on the performance of the model, in our case, measured by the A_{mean} (see Table 6.1). Specifically, at each instant n, the PH method computes the CUSUM (C_n) of the deviations between the current performance value (A_{mean_n}) and the mean of the performance values obtained in all the previous periodic checks (see Equations 1 and 2). If the CUSUM of the deviations falls below a pre-defined λ threshold (see Equation 3), the PH test signals a change ($PH_n = 1$) that triggers a model update at instant n. Note that a higher tolerance value may result in a lower rate of false alarms, but also in a lower performance, as updates can be delayed. When a change is detected, the values used for the test are reset. This means that the instant n, at which the test flags the change, is set as the starting point (0) for subsequent calculations of the test.

$TPR = \frac{TP}{P}$	$TNR = \frac{TN}{N}$	$A_{mean} = \frac{TPR+TNR}{2}$

Table 6.1: Metrics used in this chapter to assess the performance of detectors. TPR = True Positive Rate. TNR = True Negative Rate.

6.5 Data Used for Retraining

The effectiveness and efficiency of retraining depends not only on the frequency with which the models are updated, but also on the data used for this purpose. In this section, we analyze state-of-the-art methods such as fixed-size sliding windows and active learning methods such as uncertainty sampling and contrastive learning

OOD techniques. These methods have been chosen due to their popularity and also taking into account the availability of code. Finally, we present a problem-specific sample selection strategy.

6.5.1 Sliding Windows

In this approach, a fixed-size sliding window is used to select the m most recent instances for retraining. The window moves forward whenever new data becomes available, and instances that fall within the window are stored and subsequently used for retraining, while older apps are discarded. We depict this policy in Figure 6.3. Its implementation is straightforward, but it may have some drawbacks. First, it assumes that the m most recent apps are sufficiently representative to generate a good model, which may not always be true: behaviors of discarded apps may reappear later. Secondly, this method does not consider the characteristics of the instances within the window. A common feature of the Android app environment is the presence of majority groups, that is, apps that are nearly identical and appear in large quantities. Not considering the characteristics of the last m apps might result in datasets where some types of apps are over-represented while others are largely underrepresented, thus leading to biased models which ignore the minority samples [108].

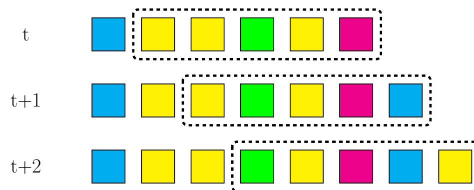


Fig. 6.3: Sliding window (dotted line) of size $m = 5$ which is used to train the model at each instant. Colored cubes represent apps with different behavioral patterns and whose predominance may vary over time. A model trained at time t may be biased towards the “yellow” behavior, while being unable to recognize the “blue” behavior.

6.5.2 Uncertainty Sampling

Uncertainty sampling is a technique commonly proposed in the active learning literature to reduce labeling efforts and improve the learning ability of models [313]. The method measures the reliability or uncertainty of the decisions provided by models for samples. The degree of uncertainty of a sample is computed as the complementary of the absolute difference of the class (goodware and malware) probabilities returned by a model [3]. Since low confident decisions for samples result in similar probability values for both classes, the uncertainty value will be high (close to 1), whereas samples where one class probability dominates the other

will obtain low uncertainty values close to 0. The method assumes that samples with the highest uncertainty are the most representative of changes and better candidates for learning a new model. Therefore, two alternative criteria can be used to build the training dataset: (1) set a fixed number n of samples to select and, (2) set a minimum uncertainty value to select the samples. Finally, the selected samples are added to the samples used for the previous retraining period and a new model is built with all this data.

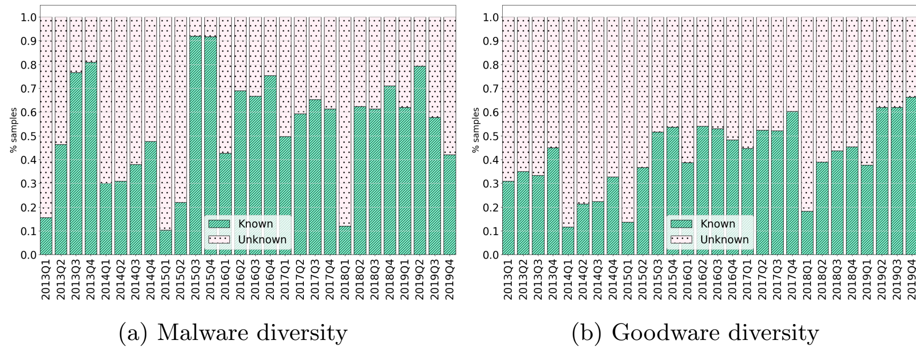
6.5.3 Contrastive Learning OOD

More advanced sampling mechanisms use contrastive learning (CL) schemes that rely on an encoder-decoder architecture to identify drifting samples and incorporate them to the training set in order to capture the new dynamics in the data. The CL model is trained to generate similar embedded representations (or embeddings) for samples of a same class or malware family, whereas the embeddings for samples of different classes (malware vs goodware) or malware families are dissimilar. CL methods then, measure the dissimilarity between the embeddings of new samples and the embeddings of the training data; the idea is that new samples whose representation differs a lot from the training data are drifting samples. Similar to uncertainty sampling methods, a ranking is constructed based on this dissimilarity measure. This is the case for CADE [314] and HICL [62]. Afterwards, samples are selected by: (1) taking the n most dissimilar samples, or (2) setting a threshold over the dissimilarity measure. Finally, the selected samples are appended to the training samples of the previous retraining period.

6.5.4 Problem Specific Sample Selection

As mentioned previously, in the context of Android malware detection, apps with some specific features may be more prevalent than others. Recurring malware that fades away and resurfaces is also a reality. To exemplify this, Figure 6.4 depicts the distribution of goodware and malware into known and unknown behaviors for every quarter between January of 2013 and December of 2019. In this context, an app behavior is represented by a particular set of API call frequencies extracted from its code. We assume that apps with similar behavior present equivalent API call frequencies in their code. The exact process for the computation of known and unknown behaviors is explained more in detail later in this section. Green slashed bars represent the proportion of samples on each period that contain similar behaviors to apps observed in previous periods (known), whereas grey dotted bars represent the proportion of samples whose behavior has not been observed previously (unknown). As can be seen in Figures 6.4a and 6.4b, the apparition of unknown app behaviors from one period to another confirms the existence of data drift in the Android app ecosystem, which can cause model degradation [63]. It also shows that the incidence of drift is variable (for example, differences in malware between 2015Q2 and 2015Q3). In this regard, goodware tends to present more novel patterns over time, whereas malware frequently exhibits more known behaviors that have been

observed in preceding periods. Indeed, the fluctuations observed for the malware follow a common infection pattern. Each time a new form of infection emerges, the apps (samples) exploiting this method will be initially classified as unknown (see, for example, 2015Q1). Then, the infection mechanism becomes popular as new malware apps use it. This is shown, for example, by the increase of known groups in the subsequent periods to 2015Q1. This popularity keeps increasing until the infection pattern is detected and a new form or a variation of the original exploitation mechanism is developed.



(a) Malware diversity (b) Goodware diversity

Fig. 6.4: Diversity of the dataset throughout the evaluation period (2013-2019). The bars represent the percentage of apps captured in the indicated period that are very similar to apps that have already appeared in previous periods ("Known"), or apps that exhibit new behaviors ("Unknown").

The over-representation of malware with known behaviors during most periods can lead to biased detectors when retraining ML models, as algorithms are designed to optimize performance metrics and may focus solely on these majority groups [331]. Hence, using all the data for training can hinder the ability of detectors to accurately distinguish minority (unknown or new) malware. With the aim of improving the effectiveness of the adaptation mechanism and producing more reliable malware detectors, we propose the use of an ad-hoc sample selection approach for Android malware detection. This technique ensures that the retraining data is diverse and informative [209]. It involves filtering out uninformative or duplicated apps, controlling the size of the dataset, and reducing the labeling costs and training complexity of ML algorithms.

Particularly, in this chapter we propose a sample selection method using the continuous clustering process described in Chapter 4 and initially proposed in [236]. For this algorithm, and based on previous findings, we represent the apps as a vector of frequencies of their Android API calls. The process works in two phases: the calibration phase, where samples in the training set are grouped into known behavioral clusters, and the online phase, where incoming samples are assigned into known or unknown clusters to perform sample selection.

The objective of the calibration phase is to find the different behavioral groups present in the training data, for both malware and goodware. To do so, the apps in the training set are chronologically ordered by their publication date and sequentially assigned to their closest cluster. This assignment is only performed if the sample lies within a predefined ϵ radius from the cluster's representative; otherwise, a new cluster is created with the sample as the representative. We assume that samples within a group contain similar code patterns and thus, that each cluster represents a particular behavioral pattern. Note that cluster's representatives are maintained throughout the process. The Euclidean distance is used to measure the similarity between every pair of samples. Once all the apps are clustered, we label the clusters as goodware or malware according to the class label of the representative app of that cluster. Within this calibration phase we compute the average number of apps in all the behavioral clusters found (k). Then, we only keep the most recent k components (apps) from each cluster. In this way, we try to keep the training set both small (keeping only a few samples of a given behavior) and diverse (keeping samples of all the different behaviors detected).

During the online phase (see Algorithm 1), at each retraining period, the method assigns each new incoming sample to its closest cluster if it meets the admission condition (the sample is within the ϵ radius of the representative). If the cluster already contains k samples, the oldest sample in the cluster is replaced by the new one. This process can be seen as a multi-window approach in which a sliding window of size k is maintained for each of the behavioral clusters. If a sample cannot be associated with an existing cluster, a new cluster is created with that sample as its representative. Once all the new instances have been clustered, we compute the isolation level of clusters as the average Euclidean distance between the cluster representative and the representatives of other clusters. To accommodate different labeling budgets, we use the parameter l_b to select the representatives of the most isolated new behavioral clusters, where l_b represents the labeling budget. Then, the retraining dataset is constructed by appending the most recent k samples from each labeled cluster to the samples used in the previous retraining period. Note that the apps assigned to a cluster are automatically labeled with the class label of the cluster representative, reducing the need to label many apps.

6.6 Experimental Framework

This section describes the experimental set-up and the methodology followed to evaluate the different adaptation mechanisms analyzed in this chapter.

6.6.1 Dataset

In our experiments, we use the dataset presented in Section 4.5.1 of this dissertation. We split this dataset into two separate subsets: one for training and one for evaluation. The training dataset consists of 100 monthly samples of each class,

Algorithm 1 Problem-specific sample selection process

```
1: X ← {x_i | i = 1, ..., N}                ▷ Set of unlabeled samples
2: C ← {c_t | t = 1, ..., T}                ▷ Set of existing cluster centroids
3: Q ← {}                                   ▷ Set of new cluster representatives
4: W ← {}                                   ▷ Set of new cluster weights
5: for i = 1, ..., N do                     ▷ Loop over unlabeled samples
6:     mind ← dist(x_i, c_1)
7:     minid ← 1
8:     for t = 2, ..., T do                 ▷ Find the closest existing cluster
9:         if dist(x_i, c_t) < mind then
10:            mind ← dist(x_i, c_t)
11:            minid ← t
12:    if mind < ϵ then
13:        x_i → c_minid                    ▷ Add x_i to an existing cluster
14:    else
15:        C ← C ∪ {x_i}                    ▷ Create a new cluster
16:        Q ← Q ∪ {x_i}
17: for j = 1, ..., |Q| do
18:     w_j ← avg(dist(q_j, Q))             ▷ Isolation level of clusters
19: R ← rank(Q, W)                          ▷ Rank clusters based on weights
20: label(R, min(|Q|, l_b))                 ▷ Label the selected clusters
```

goodware and malware, between January 2012 and December 2012. Note that malware detection on Android is a highly unbalanced problem where malware actually accounts for about 10% of the apps [232]. However, the training dataset is compiled offline and, thus, can be constructed using an unrealistically balanced ratio between the two classes. Contrarily, in order to mimic a real situation, the remaining 7 years of data (from January 2013 to December 2019), used for model evaluation purposes, will consists of 10 malware and 100 goodware samples per month over this period, which are obtained by randomly sampling apps from the original dataset.

6.6.2 Batch Malware Detectors

For the purpose of this chapter, we rely on three state-of-the-art malware detectors, Drebin, DroidDet and MaMaDroid, that constitute the best performing batch Android malware detectors published to date as evidenced in Chapter 4. These detectors were not originally conceived to cope with concept drift and they all rely on features extracted through static analysis of APK files to represent the apps, and on batch models for detection. In this section we briefly describe their detection mechanisms:

Drebin [24] uses a full set of features extracted from APKs, including hardware components, permissions, app components, intent filters, strings, and a restricted set of API calls. It uses a linear SVM model fed with this data to perform malware detection.

DroidDet [334] relies on data obtained exclusively from the app code to detect malware. More specifically, it uses a filtered set of requested and required permissions, intent filters and API calls. After the first extraction of all possible values, the relevance of the features is calculated to eliminate those that are not informative. The most relevant features are finally used for model generation using the RotationForest algorithm.

MaMaDroid [222] constructs a Markov chain of the API calls found in the app code. The Markov chain represents the transition frequency between each API pair. Actually, the package to which an API call belongs is used as a higher level abstraction to reduce the number of final features. The RandomForest algorithm is used to identify malware with this information.

6.6.3 Parameter Settings

The baseline (original) detectors have been trained using the default parameters reported in their respective works. For all configurations, the evaluation and (possible) retraining process is set to quarterly intervals. This choice is a compromise between obtaining an adequate visualization of the results but restricting the number of new models, since training the models has a significant experimental cost. In addition, for the change detection method, after preliminary experiments, we set the λ threshold for the PH test to 0.02 based on preliminary results, as a threshold between detection performance and the number of retraining steps.

In relation to the methods proposed for selecting the data used for retraining, sliding windows of 100, 1000, and 2000 apps are considered in the experimentation. For the problem-specific sample selection strategy, based on preliminary tests, we set the ϵ radius (the maximum distance allowed to consider any sample as part of an existing cluster) to 0.01. The k value (the average number of apps in a cluster), has been calculated in the calibration phase, taking a value of 2. For CL methods (CADE [314] and HICL [62]), we use the original implementation and the parameters that reported the best results in their respective papers[1][2]. Additionally, uncertainty and CL methods require setting a criterion for selecting the samples to be labeled and appended to the retraining dataset. In this sense we fix parameter n, the number of most uncertain samples in each retraining round in order to experiment with different labeling budgets. Specifically, we select 70% of the incoming samples, which is equivalent to the number of labels required by the problem-specific sample selection method with $\epsilon = 0.01$, and 45% and 15%, which are the test values considered in the HICL paper.

6.6.4 Evaluation Framework and Metrics

First, all models are trained offline (batch) using the apps in the training dataset (balanced and with data from Jan. 2012 to Dec. 2012). For evaluation purposes, we

[1] `https://github.com/whyisyoung/CADE`
[2] `https://github.com/wagner-group/active-learning`

consider non-overlapping windows of three-month periods. Therefore, the evaluation dataset is divided into 28 time-ordered subsets, each one covering one quarter.

For the evaluation of the original version of the detectors (pure batch scenario without retraining), the model used is always the same, i.e., the one obtained in the offline phase. For those scenarios incorporating concept drift management approaches, model update procedures are subsequently carried out with a subset of recent apps. Note that we assume that, when a model is updated, the true labels of the samples used to train the new model are known. As the incoming data are chronologically sorted, we can evaluate the degree of concept drift, as well as the effectiveness of the measures implemented to address it. This approach is common in the concept drift literature [102].

In two separate experiments we analyze and compare the effect of: (1) the policies to trigger the updates, and (2) the data used for retraining. In the first experiment, periodic vs. change detection mechanisms are compared. In this experiment, the dataset used for training the models grows in each retraining round since all incoming samples are incorporated to the dataset for retraining. In the second experiment, where the different data selection mechanisms are studied, the models are retrained at each three-month period with the corresponding selection of data (windows of fixed size, uncertainty samples, OOD samples or cluster components). As a baseline for this second experiment, we also consider the model retrained periodically every quarter using all available data.

In each retraining round, for all the possible configurations using the problem-specific sample selection method, goodware is downsampled to reach a balanced ratio between the malware/goodware classes. This reduces the amount of data and avoids imbalance when retraining. Specifically, when the training dataset is constructed using the problem-specific sample selection method, once the clustering has been carried out and the goodware and malware samples are obtained, 90% of the goodware samples are removed. Note that this is only done for training, whereas for evaluation the original unbalanced data is used.

Finally, in all the experiments and for each model, we measure its performance as the average of the TPR and TNR, known as the A_{mean} value (see Table 6.1). The A_{mean} is a popular performance metric in the ML literature for unbalanced scenarios and, contrary to the F1 score, the A_{mean} considers and equally weights the accuracy of models on both positive (malware) and negative (goodware) samples.

6.7 Experimental Results

This section shows the results of the different retraining configurations tested for the state-of-the-art malware detection models: Drebin, DroidDet and MaMaDroid. Code implementations for all these mechanisms are available in our GitLab repository[3].

[3] `https://gitlab.com/serralba/concept_drift`

6.7.1 Analysis of the Effect of the Retraining Frequency

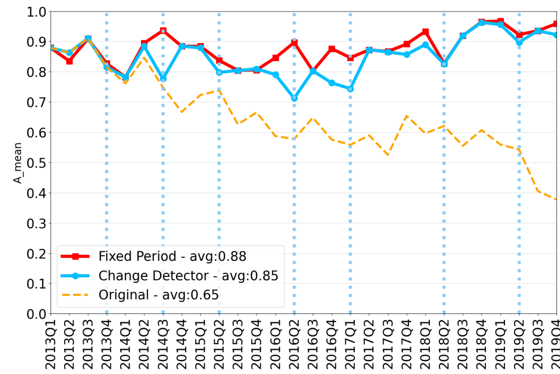
The results when retraining the detectors at fixed periods and with change detection are shown in Figure 6.5. The lines in the figures represent the A_{mean} performance of the models over the evaluation period. In particular, the red lines show the performance of the detectors when a periodic retraining approach is applied. The blue lines represent the performance of detectors implementing the change detection mechanism based on the PH test. The vertical dotted blue lines represent the points at which the PH test has triggered a drift alarm and, thus, a retraining and model replacement operation has been performed. For comparison purposes, we also include the performance of the (original) batch model which is trained only once, at the beginning. This is represented by a dashed orange line.

As can be seen, the orange lines show a decreasing trend over time for all models, confirming the existence of concept drift. The benefits of using retraining as an adaptation mechanism to counteract the effect of concept drift in batch malware detectors are readily apparent from the figures. For all adaptive solutions, the performance of the models is kept stable over time. In fact, the retraining variants of DroidDet (see Figure 6.5b) show an overall performance improvement with respect to the static version of 15%, while for Drebin and MaMaDroid this performance increases 23% and 16%, respectively (see Figures 6.5a and 6.5c).

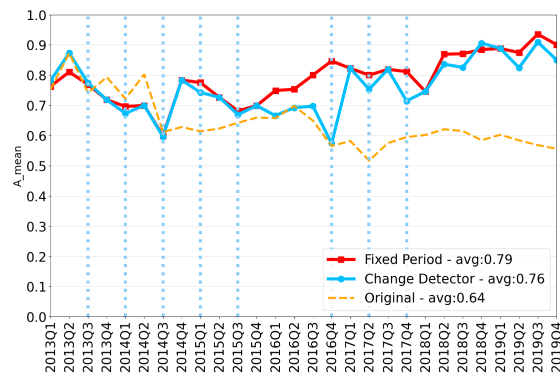
Overall, when comparing the two retraining configurations, the figures indicate that applying a change detection mechanism has a minimal cost in performance (A_{mean}), with an average reduction of 2.3% for all detectors. Conversely, the change detection method requires a much smaller number of retraining operations compared to retraining at fixed periods. In fact, as can be seen, the change detector successfully triggers a drift alarm when the performance of the detectors decreases. For DroidDet, eight rounds of retraining and model replacement are required, as shown by the blue dotted lines in Figure 6.5b, which contrasts with the 28 operations performed with fixed-period retraining. With equivalent detection performance indicators, only seven drift alarms are triggered in MaMaDroid (see Figure 6.5c) and Drebin (see Figure 6.5a).

6.7.2 Analysis of the Effect of the Retraining Data

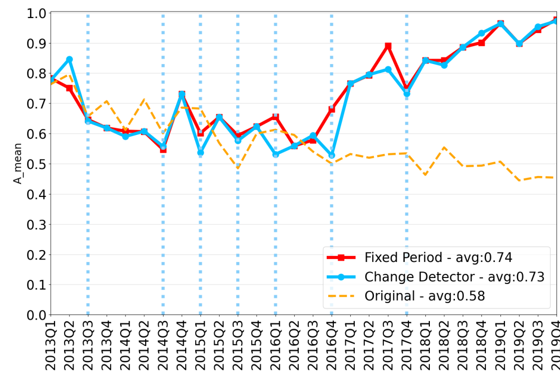
Table 6.2 displays the average performance (A_{mean}) of the detectors when using various data management policies during periodic model retraining. It is noteworthy that, in most cases, the application of a data management policy results in performance values very similar to the baseline (which utilizes all available data), with a slight decrease in performance observed in some instances as the number of samples used is reduced. Among the tested configurations, the problem-specific sample selection method, with a labeling budget of 70% of incoming samples, is the only one that outperforms the baseline by an average of 2%. Under the same labeling constraints of 70%, HICL and uncertainty methods achieve performance levels similar to the baseline configuration. Generally, except for CADE, the results do not exhibit significant differences among the active learning methods. The use of



(a) Drebin



(b) DroidDet



(c) MaMaDroid

Fig. 6.5: Performance evolution of malware detectors for the period 2013-2019, for different policies to trigger retraining. Dotted, blue vertical lines indicate a model change triggered by the concept drfit detection mechanism.

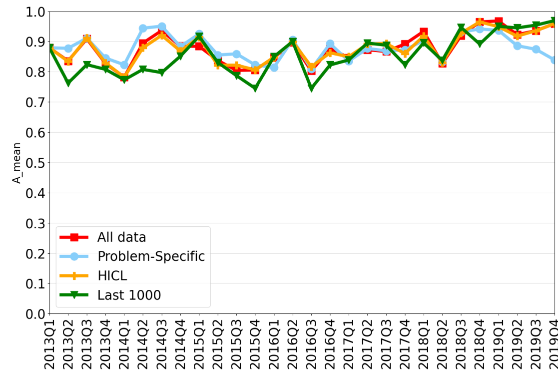
	Drebin	DroidDet	MaMaDroid	Avg. Perf.	% Labels Req.
All data	0.88	0.79	0.74	0.80	100%
Last 2000	0.87	0.79	0.72	0.79	100%
Last 1000	0.85	0.80	0.72	0.79	100%
Last 100	0.75	0.69	0.68	0.70	30%
Problem-Specific	0.88	0.78	0.82	0.82	70%
CADE [314]	0.83	0.76	0.74	0.77	70%
HICL [62]	0.88	0.78	0.75	0.80	70%
Uncertainty [3]	0.86	0.79	0.75	0.80	70%
Problem-Specific	0.86	0.76	0.78	0.80	45%
CADE [314]	0.77	0.74	0.72	0.74	45%
HICL [62]	0.87	0.77	0.75	0.80	45%
Uncertainty [3]	0.83	0.80	0.74	0.79	45%
Problem-Specific	0.84	0.73	0.76	0.78	15%
CADE [314]	0.69	0.75	0.70	0.71	15%
HICL [62]	0.86	0.75	0.74	0.78	15%
Uncertainty [3]	0.83	0.72	0.73	0.76	15%

Table 6.2: Average A_{mean} performance throughout the evaluation period for different sample selection policies using fixed period retraining. The right column refers to the percentage of samples that need to be labelled for retraining.

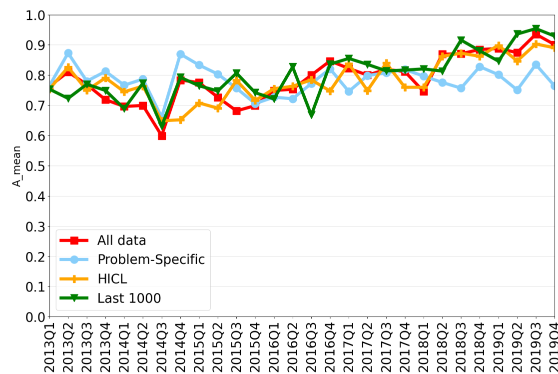
relatively small datasets with a labeling budget of 45% is sufficient to maintain performance indicators at levels very similar to, or even better than, the configuration that uses all available data.

Figure 6.6 shows the results for each individual detector over the entire evaluation period. For the sake of clarity, we have only selected the best sliding window policy, the best contrastive learning OOD method (HICL), and the problem-specific sample selection method. The red lines represent the baseline, which uses all available data for retraining; the green lines represent the performance using a sliding window of size 1000 for retraining; the orange lines represent the performance using the HICL method with a labeling budget of 70%; and the blue lines show the performance of the problem-specific sample selection mechanism with 70% of the labeling budget. Using active learning mechanisms to select samples for retraining results in improved performance values relative to the baseline and fixed-size sliding window configuration for all methods except DroidDet, with the problem-specific method yielding slightly better A_{mean} values than the HICL method in most evaluation rounds.

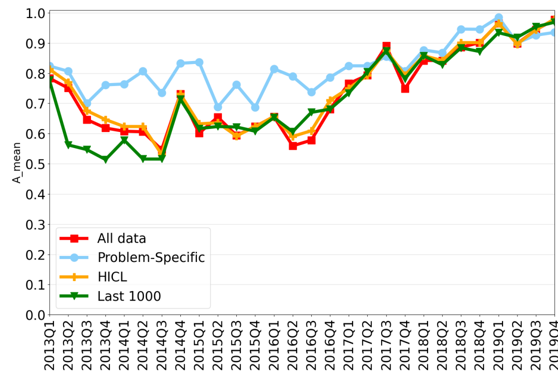
Beyond detection performance, the effort required to label the samples used in each round of retraining is also an important factor to measure the efficiency. Considering that a total of 330 apps arrive in each retraining round (300 goodware and 30 malware), the labeling requirements for the strategies “Last 1000” and “Last 2000” are similar to those of the baseline method, as they involve labeling all new arriving samples before retraining. In contrast, the “Last 100” strategy requires labeling only about 30% of the incoming samples in each evaluation round. Active learning methods require labeling only 45% of the incoming samples to obtain equivalent performance values to the baseline and the last 1000 and 2000 sliding window policies. With a lower labeling budget, the problem-specific and HICL



(a) Drebin



(b) DroidDet



(c) MaMaDroid

Fig. 6.6: Performance evolution of malware detectors with periodic retraining for the period 2013-2019. Red lines represent the performance when using all the data available, i.e., the baseline. Green, blue and orange lines represent respectively the performance of models when retrained with the 1000 most recent samples, using the problem-specific strategy and with the HICL selection methods with labeling budgets of 70%.

methods obtain very similar performance on average. These results demonstrate how detection models benefit from the use of incremental clustering to label samples and reduce the size of the training data. As a potential drawback, note that this process can lead to labelling errors. In this regard, our experiments showed that only 0.05% of the samples are mislabeled by the method, demonstrating to be insufficient to negatively impact the detection ability of ML algorithms.

6.8 Analysis of the Combined Effect of Change Detection and Sample Selection Methods

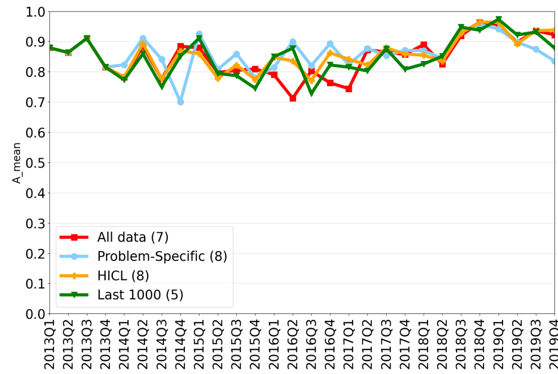
When change detection and sample selection policies are combined for retraining (see Table 6.3), a slight decrease in performance is obtained with respect to retraining at fixed periods (see Table 6.2) for all sample management policies. Again, as can be seen in the Figure 6.7, the best approach is the problem-specific method that uses a budget of samples to label of 70%, but other active learning configurations are not far behind. The worst performing approach is the one using the last 100 samples. In terms of the number of model updates, all configurations required a similar number of retraining operations (between 4 and 10) which are far less than using periodic retraining (28).

	Drebin	**DroidDet**	**MaMaDroid**	**Avg. Perf**	**Updates**
All data	0.85	0.76	0.72	0.77	7, 8, 7
Last 2000	0.86	0.75	0.66	0.75	8, 8, 4
Last 1000	0.85	0.80	0.69	0.78	5, 8, 4
Last 100	0.74	0.68	0.52	0.64	5, 7, 1
Problem-Specific	0.86	0.78	0.79	0.81	8, 7, 7
CADE	0.79	0.74	0.72	0.75	7, 7, 7
HICL	0.86	0.77	0.73	0.78	8, 10, 8
Uncertainty	0.84	0.77	0.72	0.77	8, 9, 8

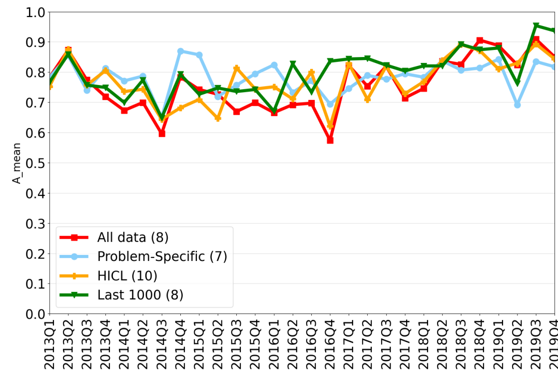
Table 6.3: Average A_{mean} performance throughout the evaluation period for different sample selection policies and change detection. Column "Updates" represents the number of changes detected by the PH test method for each of the evaluated detectors

6.9 Conclusions

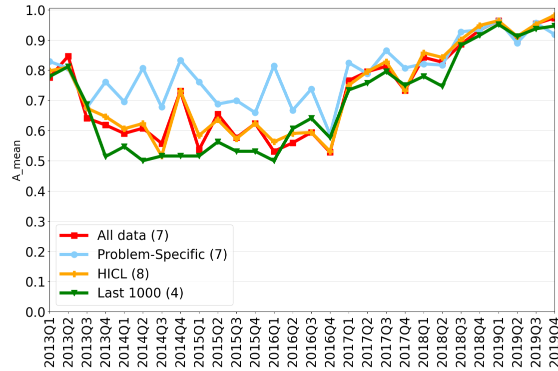
In this chapter, we have shown that retraining is an effective mechanism for dealing with concept drift in batch Android malware detectors, it being straightforward to incorporate into existing detectors without modifying their design. Specifically, our experiments show that this update mechanism helps maintain high detection rates, with an average performance improvement of 20% compared to the original



(a) Drebin



(b) DroidDet



(c) MaMaDroid

Fig. 6.7: Performance evolution of detectors with change detection for the period 2013-2019. Numbers between parentheses depict how many retraining and model replacement rounds are triggered by the change detection method to keep the effectiveness of detectors over time.

versions of the detectors. Regarding the two retraining alternatives tested, there are no significant performance differences between periodic retraining and the PH-based change detection approaches. However, using a supervision mechanism based on the PH test showed to decrease the number of retraining rounds by 75% on average, dramatically reducing the computational effort required to keep model performance over time.

Additionally, we have demonstrated that the sample selection strategy used for retraining also influences the success of detectors. Employing a sample selection policy instead of using all available data for retraining reduces the cost of model generation since the complexity of machine learning algorithms is highly dependent on the size of the training data [128]. In this sense, sliding window policies, such as selecting the last 1000 and 2000 samples, are simple and help reduce retraining complexity even if they require similar labeling efforts and performance values than the baseline. Going one step ahead, the benefits of using active learning techniques, such as uncertainty sampling, HICL, or the problem-specific sample selection method, are undeniable. These techniques result in better detection performance for models and, additionally, they require lower labeling effort for retraining. Among the active learning methods, the proposed problem-specific strategy exhibited minimal performance degradation under widened labeling constraints compared to other alternatives.

In general, the choice of a specific sample selection strategy and retraining policy will depend on the requirements of the target scenario for the detector. Change detection is a suitable method in most scenarios, especially in cases where the cost of generating models is high. One advantage over periodic retraining is that it requires fewer retraining operations to keep models up-to-date, consequently reducing the need for labeling new samples. Labeling new samples is often costly, and in many cases, it is performed manually by human experts. In this context, the use of a sample selection mechanism is also desirable. Larger sliding window sizes need labeling all incoming data, which may not be feasible, especially in online scenarios. Shorter windows, on the other hand, lead to rapid forgetting and can cause model overfitting. Most active learning approaches such as HICL, problem-specific sample selection, or uncertainty sampling are particularly useful because they can reduce the number of samples to be labeled without compromising detection performance. In addition, they do not include a forgetting mechanism, which helps mitigate the impact of recurring app behavior. It should be noted, however, that CADE and HICL are more costly because they require the generation of a new CL model for sample selection at each retraining step, whereas the cost of the problem-specific sample selection method can be considered negligible because clusters are updated incrementally, involving a one-pass process over the data.

When combining both the change detection and the sample selection methods, the problem-specific approach is the only that demonstrated to improve performance with respect to that obtained by periodically retraining models with all available data. In view of this, it is clear that the benefits of using the proposed change detection and sample selection methods are not only related to performance, since they also involve reduced labeling and model generation efforts.

Part III

Summary of Main Findings

7

Conclusions and Future work

This chapter summarizes the main contributions and conclusions of this dissertation. In addition to the specific conclusions introduced at the end of each of the previous chapters, we discuss here a number of general directions for further research.

7.1 Contributions

In our rapidly evolving technology landscape, malicious actors continually seek new ways to infiltrate information systems. In response, system protection strategies have been adapted by employing advanced technologies and intelligent methods, primarily through AI techniques such as ML algorithms. This transformation has led to impressive results in solutions for Network Intrusion Detection Systems (NIDS) and Android malware detection over the past decade, achieving detection rates exceeding 90% and false positive rates below 5%. Despite these remarkable achievements, the deployment of highly effective solutions in real-world scenarios remains a challenge. This dissertation aimed to analyze and propose practical solutions to two critical information security problems: Network Intrusion Detection and Android Malware Detection, utilizing the capabilities of artificial intelligence algorithms.

Each of the chapters of this dissertation ended with some partial conclusions. The main issues and conclusions of the research described in this dissertation are elaborated in the following paragraphs.

In Chapter 2, we have shown how a large number of papers have focused on the use of ML algorithms for the detection of network attacks with either raw packet captures or flow data. However, aspects such as the location of the network probes, the maximum throughput allowed by the system, and the attacks that can be detected are often inadequately documented or overlooked. Our study determined that the majority of these research papers relied on datasets that do not accurately represent real-world networks. These datasets frequently lack traffic samples for

protocols that employ encryption, such as HTTPS and IPsec, which are commonplace in modern networks. Additionally, many of the datasets feature short-duration captures that fail to capture the true dynamics and evolution of network traffic.

Throughout our analysis, we followed the KDD process as a guiding framework to assess the validity and feasibility of techniques employed in NIDS proposals. Our findings highlighted the shortcomings of proposals relying on NIDS datasets, showing their inadequate design that often leads to unrealistic performance values, creating an overly optimistic picture of their effectiveness. Additionally, we brought to light a critical issue within the field, i.e., the lack of reproducibility. In this regard, many proposals omit essential details, making it impossible for others to reproduce their work.

In terms of detection tasks, the work carried out has resulted in a new taxonomy that makes it possible to identify NIDS between the so-called misuse-based systems and anomaly-based types. It has been observed that most NIDS focus on misuse detection, using batch-based supervised learning algorithms. In contrast, only a limited subset of proposals explore anomaly detection using weakly or semi-supervised approaches, aiming to exploit the potential of these algorithms to identify unknown malicious patterns without the need to use extensive labeled data for training.

Regarding the evaluation procedures used in NIDS, our analises reveal that the management of the so-called concept drift is only taken into account in a small portion of the proposals. Furthermore, we conclude that current complexity analyses are inadequate because, in order to comprehensively assess computational costs, evaluations should encompass all the stages involved in detection, from the capture and preprocessing of network traces to the final results of the decision under analysis of such network traffic. All of these facts have led us to determine that proposals based on unrealistic datasets exhibit inadequate design, often accompanied by deficient evaluations that result in unrealistic performance values and create an overly optimistic image of their effectiveness.

In the Android malware detection area, and similarly to the NIDS area, detection proposals that leverage static analysis and ML algorithms are based on datasets containing both malware and goodware samples. Firstly, we have explained the reasons behind the prevalence of proposals that built their own datasets and experimental setups. Throughout Chapter 4, we have identified and evaluated five factors that are present in realistic environments and may determine the reliability (and realism) of current solutions. These factors include the presence of duplicates, the exclusion of greyware, labeling biases, the stationarity assumption, and evasion attempts. We conducted evaluations on ten highly influential works in the field, which we were able to reproduce. Our findings consistently revealed that the exceptional performance values reported in the original papers could not be sustained under varying configurations. This underscores the critical importance of considering these factors for more realistic evaluations.

Aiming to tackle some of the limitations evidenced in Chapter 4, in Chapter 5 we analyzed how classical evasion techniques such as obfuscation can affect static analysis features used for malware detection. This study had an intrinsic second purpose: evaluate how useful is obfuscation to bypass detectors leveraging static

analysis. The picture represented in our experiments evidenced that the level of impact of obfuscation on different families of static analysis features varies among obfuscation techniques and tools. Contrary to what it is typically assumed, we demonstrated that feature persistence is not a good indicator of the robustness of features. In fact, some families of persistent features led to model malfunction when specific types of obfuscation are applied to samples. In this regard, we proposed to use feature insensitivity, a measure based on the decisions of models for non-obfuscated and obfuscated apps, which demonstrated to be a better indicator of the robustness of features. As a result of this study, we identified the most insensitive families of features (Permissions, Strings and API functions), and proposed a detector making use of these features that outperformed the state-of-the-art under the zero-knowledge assumption (without any information about obfuscation).

Also, the evaluations performed in Chapter 4 identified that the best malware detection methods to date are Drebin (presented in 2014), DroidDet (presented on 2018) and MaMaDroid (presented in 2019). These methods leverage batch learning algorithms that cannot cope with app evolution. In fact, through Chapter 6 we showed how these three detectors rapidly loose their accuracy in real deployments due to concept drift. Our evaluations demonstrated that the application of retraining in combination with a drift detection mechanism and a buffer management method is a valid solution to efficiently adapt existing batch ML malware detectors for Android. In particular, we showed that leveraging the Page-Hinkley test to detect changes in the data distribution helps to maintain the performance of detectors over time and has minimal cost because the number of retraining steps is reduced. Additionally, uncertainty sampling and problem-specific sample selection approaches showed limited impact on the performance of models but, instead, presented a higher reduction on labeling requirements.

As a final conclusion, the lack of standard evaluation procedures, the assumption of unrealistic scenarios, the omission of key implementation details, and the absence of working codes are particularly prevalent in information security research. Each of these issues makes it difficult to reproduce proposals, which slows down the emergence of realistic security solutions based on KDD and ML/AI algorithms.

7.2 Future work

The work embodied in this thesis has served to identify several open lines of research that merit further work. Below we list some of them, both in the field of NIDS and Android malware detection.

The analysis of the NIDS literature carried out in Part I of this dissertation evidenced the following open research questions:

- The realism of current NIDS databases is not frequently analyzed and more efforts should be put to evaluate how the available datasets can be improved, helping on the proposal of novel datasets that include the latest trends in network traffic.

- A set of common NIDS evaluation guidelines is required, allowing to fairly compare different products and proposals.
- Efficient capture and decryption mechanisms are required to extract the plaintext version of encrypted payloads with low overheads. A step towards a solution may be the use of probes installed at the hosts (that is, once the data is received and decrypted), but at the cost of increasing network traffic (payload still has to be sent to the NIDS for analysis) and losing some early detection capabilities. This aspect algo brings the opportunity for novel learning strategies such as Federated Learning [39].
- We also think that more sophisticated variables could be useful to increase detection performance, widen the spectrum of identified attacks, harden the security of a NIDS and identify evasion attempts. For example, features that summarize the additional traffic activity caused by a network connection (number of different service requests, such as DNS, ARP, etc.), together with indicators of causal relationships, may also be useful for the detection of contextual patterns of network attacks. In general, the extraction and use of additional SFD and MFD features needs to be further explored, specially for dealing with different kinds of attacks and evasion attempts [135].
- It has been proven that NIDS proposals have to shift towards incremental solutions since non-stationarity is a characteristic of communication networks [293]. This involves an extensive analysis of the most adequate detection strategies, examining the advantages, shortcomings and use-cases of misuse, anomaly and hybrid approaches, as well as of unsupervised and supervised ML algorithms. This also would help delimit the scope of some detection mechanisms and help on the emergence of detection alternatives.
- Performing early detection of malicious traffic with models that exploit temporal patterns and dependencies among different communication protocols could be of interest since it can limit attack consequences. For example, the analysis of ARP flows could be used to detect some man-in-the-middle attacks at an early stage.

In the Android malware detection area, the experimental framework and contributions made to deal with concept drift and obfuscation presented along Part II evidenced the following potential research lines:

- We limited our study to static analysis detectors for simplicity since dynamic analysis detectors involve a large number of implementation and configuration decisions when performing app analysis. Future work should strive to propose reproducible and standardized evaluation procedures, as we have done for static analysis works, in dynamic analysis approaches.
- Labeling carried out leveraging the number of detections in VirusTotal (VTD) presents some shortcomings. Differences in anti-virus tools are present. These are even more noticeable when obtaining family labels from malware, since anti-virus firms do not share a common nomenclature. In this regard, despite that some useful tools are available for agreeing on family labels from VirusTotal reports [261], we believe that there is still a lot of work to be done. Additional

information such as the details included in the VirusTotal reports can be used to support more sophisticated labeling techniques, identifying particular functionalities like advertising, bundling, etc. which could provide valuable information to the user about a given app. This would help to create a taxonomy of the type of apps present in the Android ecosystem, and to reach consensus on what is goodware or malware. It would also help to determine the usefulness of a third category of "potentially unwanted apps" [115] and its consideration as a third class in the data. Furthermore, to avoid the use of fixed thresholds over the VTD for labeling, crowd learning should be explored further [154]. As an alternative to binary or three-class classifiers, new detection proposals could explore regression models that provide a risk/maliciousness metric, multi-step learning approaches [69], unsupervised detectors that return a degree of dissimilarity with respect to the benign class [196], or semi-supervised methods that do not require fully labeled datasets [291].

- The use and design of specific ML classifiers for unbalanced scenarios is an open research line, which could be promising in Android malware detection. Some examples include cost-based classification methods, and subsampling and oversampling methods [92]. In addition, depending on the particular scenario in which the model will be deployed, and on the interest of the practitioner (for example, reducing the number of false positives), different loss functions could be used in the training phase. As alternative approaches, anomaly detection algorithms could also be considered. All these aspects remain currently unexplored.
- In retraining scenarios with sample selection approaches, more complex sliding window mechanisms could be analyzed. These may include methods adapting the size of the sliding window as a function of the distribution dynamics [44] to deal with applications that manifest themselves in different ways, e.g., periodically or recurrently. Also, sample selection approaches that aim to optimize both the size and the informativeness of the dataset that is used for retraining, for example using selective and adaptive forgetting mechanisms, can provide an advantage over existing sliding window or uncertainty mechanisms.
- Appropriate assessments of the robustness of detectors should consider all the steps involved in the malware detection process. These should include the evaluation of detectors against attacks that target the analysis phase of apps, such as sandbox detection [321] and obfuscation (as we did), or to attacks that target ML algorithms, such as those that leverage adversarial learning [325]. Essentially, novel evaluation methodologies that take into account the latest technology and attacks are needed and represent a promising future line of research.
- An interesting line of future research could analyze whether static analysis frameworks present vulnerabilities that can increase the impact of obfuscation for evading detectors. This aspect would help developers to improve static analysis tools and also, facilitate practitioners to select the most reliable ones.
- The proposal and implementation of better obfuscation strategies and tools for Android is a promising research area. Such strategies should be accompanied with evaluations to ensure that the execution of the applications is not broken.

- Richer and more robust app representations should be further explored. These may include the benefits that the integration of different static and dynamic analysis data has, as well as novel features based on the extraction of temporal execution patterns from app code.

We conclude this section with a more general recommendation for future work. The research conducted for this thesis has brought to light a troubling issue: the lack of reproducibility of proposals and evaluations of NIDS and Android malware detectors. We strongly believe that researchers need to adopt more open and transparent practices, such as providing detailed documentation, code and datasets, and using standardized evaluation methodologies.

7.3 Publications

The research work carried out during this thesis has produced the following publications in referred journals:

- **B. Molina-Coronado**, U. Mori, A. Mendiburu, J. Miguel-Alonso (2020). Survey of network intrusion detection methods from the perspective of the knowledge discovery in databases process. *IEEE Transactions on Network and Service Management*, 17(4), pp.2451-2479.
- **B. Molina-Coronado**, U. Mori, A. Mendiburu, J. Miguel-Alonso (2023). Towards a fair comparison and realistic evaluation framework of android malware detectors based on static analysis and machine learning. *Computers & Security*, 124, p.102996.
- **B. Molina-Coronado**, U. Mori, A. Mendiburu, J. Miguel-Alonso (2023). Efficient Concept Drift Handling for Batch Android Malware Detection Models. *Pervasive and Mobile Computing*, 96, p.101849.
- **B. Molina-Coronado**, A. Ruggia, U. Mori, A. Merlo, A. Mendiburu, J. Miguel-Alonso (2023). Light up that Droid! On the Effectiveness of Static Analysis Features against App Obfuscation for Android Malware Detection. *IEEE Transactions on Dependable and Secure Computing*. Submitted.

7.4 Data and Code

In spirit of open science, the datasets and codes product of this work have been made publicly available in the following repositories:

- Data and Code for the Chapter 4: `https://gitlab.com/serralba/androidmaldet_comparative`
- Data and Code for the Chapter 5: `https://gitlab.com/serralba/robustml_maldet`
- Data and Code for the Chapter 6: `https://gitlab.com/serralba/concept_drift`

References

[1] Afianian, A., Niksefat, S., Sadeghiyan, B., and Baptiste, D. (2019). Malware dynamic analysis evasion techniques: A survey. *ACM Computing Surveys (CSUR)*, 52(6):1–28.

[2] Aggarwal, C. C. (2015). *Data mining: the textbook*. Springer.

[3] Aggarwal, C. C., Kong, X., Gu, Q., Han, J., and Philip, S. Y. (2014). Active learning: A survey. In *Data classification*, pages 599–634. Chapman and Hall/CRC.

[4] Aghakhani, H., Gritti, F., Mecca, F., Lindorfer, M., Ortolani, S., Balzarotti, D., Vigna, G., and Kruegel, C. (2020). When malware is packin'heat; limits of machine learning classifiers based on static analysis features. In *Network and Distributed Systems Security (NDSS) Symposium 2020*.

[5] Ahmad, S., Lavin, A., Purdy, S., and Agha, Z. (2017). Unsupervised real-time anomaly detection for streaming data. *Neurocomputing*, 262:134–147.

[6] Ahmed, M., Mahmood, A. N., and Hu, J. (2016). A survey of network anomaly detection techniques. *Journal of Network and Computer Applications*, 60:19–31.

[7] Al-Ghossein, M., Abdessalem, T., and Barre, A. (2021). A survey on stream-based recommender systems. *ACM computing surveys (CSUR)*, 54(5):1–36.

[8] Alaa, M., Zaidan, A. A., Zaidan, B. B., Talal, M., and Kiah, M. L. M. (2017). A review of smart home applications based on internet of things. *Journal of Network and Computer Applications*, 97:48–65.

[9] Allix, K., Bissyandé, T. F., Jérome, Q., Klein, J., Le Traon, Y., et al. (2016a). Empirical assessment of machine learning-based malware detectors for android. *Empirical Software Engineering*, 21(1):183–211.

[10] Allix, K., Bissyandé, T. F., Klein, J., and Le Traon, Y. (2015). Are your training datasets yet relevant? In *International Symposium on Engineering Secure Software and Systems*, pages 51–67. Springer.

[11] Allix, K., Bissyandé, T. F., Klein, J., and Le Traon, Y. (2016b). Androzoo: Collecting millions of android apps for the research community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 468–471. IEEE.

[12] Almgren, M. and Jonsson, E. (2004). Using active learning in intrusion detection. In *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.*, pages 88–98. IEEE.

[13] Alrawashdeh, K. and Purdy, C. (2016). Toward an online anomaly intrusion detection system based on deep learning. In *Machine Learning and Applications (ICMLA), 2016 15th IEEE Int. Conf. on*, pages 195–200. IEEE.

[14] Ambusaidi, M. A., He, X., Nanda, P., and Tan, Z. (2016). Building an intrusion detection system using a filter-based feature selection algorithm. *IEEE Trans. on Computers*, 65(10):2986–2998.

[15] Amiri, F., Yousefi, M. R., Lucas, C., Shakery, A., and Yazdani, N. (2011). Mutual information-based feature selection for intrusion detection systems. *Journal of Network and Computer Applications*, 34(4):1184–1199.

[16] Angiulli, F. and Fassetti, F. (2007). Detecting distance-based outliers in streams of data. In *Proc. of the sixteenth ACM Conf. on Inform. and knowledge management*, pages 811–820. ACM.

[17] Aonzo, S., Georgiu, G. C., Verderame, L., and Merlo, A. (2020). Obfuscapk: An open-source black-box obfuscation tool for android apps. *SoftwareX*, 11:100403.

[18] Apruzzese, G., Laskov, P., Montes de Oca, E., Mallouli, W., Brdalo Rapa, L., Grammatopoulos, A. V., and Di Franco, F. (2023). The role of machine learning in cybersecurity. *Digital Threats: Research and Practice*, 4(1):1–38.

[19] Ariu, D., Tronci, R., and Giacinto, G. (2011). Hmmpayl: An intrusion detection system based on hidden markov models. *Computers & Security*, 30(4):221–241.

[20] Arlot, S. and Celisse, A. (2010). A survey of cross-validation procedures for model selection. *Statistics surveys*, 4:40–79.

[21] Arora, A., Peddoju, S. K., and Conti, M. (2019). Permpair: Android malware detection using permission pairs. *IEEE Transactions on Information Forensics and Security*, 15:1968–1982.

[22] Arp, D., Quiring, E., Pendlebury, F., Warnecke, A., Pierazzi, F., Wressnegger, C., Cavallaro, L., and Rieck, K. (2020). Dos and don'ts of machine learning in computer security. *arXiv preprint arXiv:2010.09470*.

[23] Arp, D., Quiring, E., Pendlebury, F., Warnecke, A., Pierazzi, F., Wressnegger, C., Cavallaro, L., and Rieck, K. (2022). Dos and don'ts of machine learning in computer security. In *Proc. of the USENIX Security Symposium*.

[24] Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., and Siemens, C. (2014). Drebin: Effective and explainable detection of android malware in your pocket. In *Ndss*, volume 14, pages 23–26.

[25] Arshad, S., Shah, M. A., Wahid, A., Mehmood, A., Song, H., and Yu, H. (2018). Samadroid: a novel 3-level hybrid malware detection model for android operating system. *IEEE Access*, 6:4321–4339.

[26] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. (2014). Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269.

[27] Atli, B. G., Miche, Y., Kalliola, A., Oliver, I., Holtmanns, S., and Lendasse, A. (2018). Anomaly-based intrusion detection using extreme learning machine and

aggregation of network traffic statistics in probability space. *Cognitive Computation*, 10(5):848–863.

[28] Axelsson, S. (2000). Intrusion detection systems: A survey and taxonomy. Technical report, Chalmers University of Technology.

[29] Bacci, A., Bartoli, A., Martinelli, F., Medvet, E., Mercaldo, F., and Visaggio, C. A. (2018). Impact of code obfuscation on android malware detection based on static and dynamic analysis. In *ICISSP*, pages 379–385.

[30] Baek, S., Kwon, D., Kim, J., Suh, S. C., Kim, H., and Kim, I. (2017). Unsupervised labeling for supervised anomaly detection in enterprise and cloud networks. In *2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud)*, pages 205–210. IEEE.

[31] Baena-Garcıa, M., del Campo-Ávila, J., Fidalgo, R., Bifet, A., Gavalda, R., and Morales-Bueno, R. (2006). Early drift detection method. In *Fourth international workshop on knowledge discovery from data streams*, volume 6, pages 77–86. Citeseer.

[32] Bakour, K., Ünver, H. M., and Ghanem, R. (2018). The android malware static analysis: techniques, limitations, and open challenges. In *2018 3rd International Conference on Computer Science and Engineering (UBMK)*, pages 586–593. Ieee.

[33] Ban, T., Samuel, N., Takahashi, T., and Inoue, D. (2021). Combat security alert fatigue with ai-assisted techniques. In *Cyber Security Experimentation and Test Workshop*, pages 9–16.

[34] Barbara, D., Couto, J., Jajodia, S., Popyack, L., and Wu, N. (2001a). Adam: Detecting intrusions by data mining. In *In Proc. of the IEEE Wksp. on Inform. Assurance and Security*. IEEE.

[35] Barbara, D., Wu, N., and Jajodia, S. (2001b). Detecting novel network intrusions using bayes estimators. In *Proc. of the 2001 SIAM Int. Conf. on Data Mining*, pages 1–17. SIAM.

[36] Barbero, F., Pendlebury, F., Pierazzi, F., and Cavallaro, L. (2022). Transcending transcend: Revisiting malware classification in the presence of concept drift. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 805–823. IEEE.

[37] Barreno, M., Nelson, B., Sears, R., Joseph, A. D., and Tygar, J. D. (2006). Can machine learning be secure? In *Proc. of the 2006 ACM Symposium on Inform., computer and Comm. security*, pages 16–25. ACM.

[38] Bayram, F., Ahmed, B. S., and Kassler, A. (2022). From concept drift to model degradation: An overview on performance-aware drift detectors. *Knowledge-Based Systems*, page 108632.

[39] Belenguer, A., Navaridas, J., and Pascual, J. A. (2022). A review of federated learning in intrusion detection systems for iot. *arXiv preprint arXiv:2204.12443*.

[40] Bhuyan, M. H., Bhattacharyya, D., and Kalita, J. K. (2016). A multi-step outlier-based anomaly detection approach to network-wide traffic. *Inform. Sci.*, 348:243–271.

[41] Bhuyan, M. H., Bhattacharyya, D. K., and Kalita, J. K. (2014). Network anomaly detection: methods, systems and tools. *IEEE Comm. Surveys & Tutorials*, 16(1):303–336.

[42] Bhuyan, M. H., Bhattacharyya, D. K., and Kalita, J. K. (2015). Towards generating real-life datasets for network intrusion detection. *Int. Journal of Network Security*, 17(6):683–701.

[43] Bifet, A., de Francisci Morales, G., Read, J., Holmes, G., and Pfahringer, B. (2015). Efficient online evaluation of big data stream classifiers. In *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*, pages 59–68.

[44] Bifet, A. and Gavalda, R. (2007). Learning from time-changing data with adaptive windowing. In *Proceedings of the 2007 SIAM international conference on data mining*, pages 443–448. SIAM.

[45] Bigdeli, E., Mohammadi, M., Raahemi, B., and Matwin, S. (2018). Incremental anomaly detection using two-layer cluster-based structure. *Inform. Sci.*, 429:315–331.

[46] Biggio, B., Corona, I., Maiorca, D., Nelson, B., Šrndić, N., Laskov, P., Giacinto, G., and Roli, F. (2013a). Evasion attacks against machine learning at test time. In *Joint European Conf. on machine learning and knowledge discovery in databases*, pages 387–402. Springer.

[47] Biggio, B., Fumera, G., and Roli, F. (2013b). Security evaluation of pattern classifiers under attack. *IEEE Trans. on Knowledge and Data Engineering*, 26(4):984–996.

[48] Blázquez-García, A., Conde, A., Mori, U., and Lozano, J. A. (2021). A review on outlier/anomaly detection in time series data. *ACM Computing Surveys (CSUR)*, 54(3):1–33.

[49] Bolón-Canedo, V., Sánchez-Maroño, N., and Alonso-Betanzos, A. (2009). A combination of discretization and filter methods for improving classification performance in kdd cup 99 dataset. In *Neural Networks, 2009. IJCNN 2009. Int. Joint Conf. on*, pages 359–366. IEEE.

[50] Breiman, L. (2001). Random forests. *Machine learning*, 45(1):5–32.

[51] Brown, C., Cowperthwaite, A., Hijazi, A., and Somayaji, A. (2009). Analysis of the 1999 darpa/lincoln laboratory ids evaluation data with netadhict. In *2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications*, pages 1–7. IEEE.

[52] Brutlag, J. D. (2000). Aberrant behavior detection in time series for network monitoring. In *LISA*, volume 14, pages 139–146.

[53] Buczak, A. L. and Guven, E. (2016). A survey of data mining and machine learning methods for cyber security intrusion detection. *IEEE Comm. Surveys & Tutorials*, 18(2):1153–1176.

[54] Bukac, V., Tucek, P., and Deutsch, M. (2012). Advances and challenges in standalone host-based intrusion detection systems. In *International Conference on Trust, Privacy and Security in Digital Business*, pages 105–117. Springer.

[55] Campbell, S. and Lee, J. (2011). Intrusion detection at 100g. In *SC'11: Proc. of 2011 Int. Conf. for High Perf. Computing, Networking, Storage and Analysis*, pages 1–9. IEEE.

[56] Canfora, G., De Lorenzo, A., Medvet, E., Mercaldo, F., and Visaggio, C. A. (2015). Effectiveness of opcode ngrams for detection of multi family android

malware. In *2015 10th International Conference on Availability, Reliability and Security*, pages 333–340. IEEE.

[57] Canfora, G., Mercaldo, F., and Visaggio, C. A. (2016). An hmm and structural entropy based detector for android malware: An empirical study. *Computers & Security*, 61:1–18.

[58] Carreño, A., Inza, I., and Lozano, J. A. (2019). Analyzing rare event, anomaly, novelty and outlier detection terms under the supervised classification framework. *Artificial Intelligence Review*, pages 1–20.

[59] Casas, P., Mazel, J., and Owezarski, P. (2011). Unada: Unsupervised network anomaly detection using sub-space outliers ranking. In *Int. Conf. on Research in Networking*, pages 40–51. Springer.

[60] Casas, P., Mulinka, P., and Vanerio, J. (2019). Should i (re) learn or should i go (on)?: Stream machine learning for adaptive defense against network attacks. In *Proc. of the 6th ACM Wksp. on Moving Target Defense*, pages 79–88. ACM.

[61] Chapelle, O., Scholkopf, B., and Zien, A. (2010). *Semi-Supervised Learning*. The MIT Press, 1st edition.

[62] Chen, Y., Ding, Z., and Wagner, D. (2023a). Continuous learning for android malware detection. *arXiv preprint arXiv:2302.04332*.

[63] Chen, Z., Zhang, Z., Kan, Z., Yang, L., Cortellazzi, J., Pendlebury, F., Pierazzi, F., Cavallaro, L., and Wang, G. (2023b). Is it overkill? analyzing feature-space concept drift in malware detectors. In *2023 IEEE Deep Learning Security and Privacy Workshop (DLSP). IEEE*.

[64] Claise, B. (2004). Cisco systems netflow services export version 9. RFC 3954, RFC Editor.

[65] Claise, B., Trammell, B., and Aitken, P. (2013). Specification of the ip flow information export (ipfix) protocol for the exchange of flow information. *RFC 7011 (Internet Standard), Internet Engineering Task Force*, pages 2070–1721.

[66] Cohen, J. (1960). A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46.

[67] Collberg, C. S. and Thomborson, C. (2002). Watermarking, tamper-proofing, and obfuscation-tools for software protection. *IEEE Transactions on software engineering*, 28(8):735–746.

[68] Cover, T. M. and Thomas, J. A. (2012). *Elements of information theory*. John Wiley & Sons.

[69] Daoudi, N., Allix, K., Bissyandé, T. F., and Klein, J. (2022). A two-steps approach to improve the performance of android malware detectors. *arXiv preprint arXiv:2205.08265*.

[70] Dash, M. and Liu, H. (2003). Consistency-based search in feature selection. *Artificial Intelligence*, 151(1-2):155–176.

[71] Davis, J. J. and Clark, A. J. (2011). Data preprocessing for anomaly based network intrusion detection: A review. *Computers & Security*, 30(6-7):353–375.

[72] De la Hoz, E., De La Hoz, E., Ortiz, A., Ortega, J., and Prieto, B. (2015). Pca filtering and probabilistic som for network intrusion detection. *Neurocomputing*, 164:71–81.

[73] Degioanni, L., McCanne, S., White, C. J., and Vlachos, D. S. (2015). Distributed network traffic data collection and storage. US Patent 8,971,196.

[74] Depren, O., Topallar, M., Anarim, E., and Ciliz, M. K. (2005). An intelligent intrusion detection system (ids) for anomaly and misuse detection in computer networks. *Expert systems with Applications*, 29(4):713–722.

[75] Desnos, A., Gueguen, G., and Bachmann, S. (2018). Androguard. [Online] Available: `https://androguard.readthedocs.io/en/latest/`.

[76] Developers, G. (2020). Enable multidex for apps with over 64k methods. Accessed online: November 2, 2023.

[77] Devore, J. L. (2011). *Probability and Statistics for Engineering and the Sciences*. Cengage learning.

[78] Do, Q., Martini, B., and Choo, K.-K. R. (2017). Is the data on your wearable device secure? an android wear smartwatch case study. *Software: Practice and Experience*, 47(3):391–403.

[79] Domingos, P. M. (2012). A few useful things to know about machine learning. *Commun. acm*, 55(10):78–87.

[80] Dong, S., Li, M., Diao, W., Liu, X., Liu, J., Li, Z., Xu, F., Chen, K., Wang, X., and Zhang, K. (2018). 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.

[81] Dougherty, J., Kohavi, R., and Sahami, M. (1995). Supervised and unsupervised discretization of continuous features. In *Machine Learning: Proc. of the 12th Int. Conf.*, pages 194–202. Morgan Kaufmann.

[82] Dromard, J., Roudière, G., and Owezarski, P. (2017). Online and scalable unsupervised network anomaly detection method. *IEEE Trans. on Network and Service Management*, 14(1):34–47.

[83] Duffield, N. et al. (2004). Sampling for passive internet measurement: A review. *Statistical Science*, 19(3):472–498.

[84] Durumeric, Z., Ma, Z., Springall, D., Barnes, R., Sullivan, N., Bursztein, E., Bailey, M., Halderman, J. A., and Paxson, V. (2017). The security impact of https interception. In *NDSS*. Internet Society.

[85] Early, J. P. and Brodley, C. E. (2006). Behavioral features for network anomaly detection. In *Machine Learning and Data Mining for Computer Security*, pages 107–124. Springer.

[86] Eskin, E., Arnold, A., Prerau, M., Portnoy, L., and Stolfo, S. (2002). A geometric framework for unsupervised anomaly detection. In *Applications of data mining in computer security*, pages 77–101. Springer.

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

[88] Faruki, P., Bharmal, A., Laxmi, V., Ganmoor, V., Gaur, M. S., Conti, M., and Rajarajan, M. (2014). Android security: a survey of issues, malware penetration, and defenses. *IEEE communications surveys & tutorials*, 17(2):998–1022.

[89] Fayyad, U., Piatetsky-Shapiro, G., and Smyth, P. (1996). The kdd process for extracting useful knowledge from volumes of data. *Comm. of the ACM*, 39(11):27–34.

[90] Fayyad, U. M. and Irani, K. B. (1993). Multi-interval discretization of continuous-valued attributes for classification learning. In *IJCAI*, pages 1022–1029.

[91] Feizollah, A., Anuar, N. B., Salleh, R., Suarez-Tangil, G., and Furnell, S. (2017). Androdialysis: Analysis of android intent effectiveness in malware detection. *computers & security*, 65:121–134.

[92] Fernández, A., García, S., Galar, M., Prati, R. C., Krawczyk, B., and Herrera, F. (2018). *Learning from imbalanced data sets*, volume 10. Springer.

[93] Fontugne, R., Borgnat, P., Abry, P., and Fukuda, K. (2010). Mawilab: combining diverse anomaly detectors for automated anomaly labeling and performance benchmarking. In *Proc. of the 6th Int. Conf.*, page 8. ACM.

[94] Forestiero, A. (2016). Self-organizing anomaly detection in data streams. *Inform. Sci.*, 373:321–336.

[95] Forouzan, B. A. (2007). *Cryptography & network security.* McGraw-Hill, Inc.

[96] Frankel, S. and Krishnan, S. (2011). Ip security (ipsec) and internet key exchange (ike) document roadmap. RFC 6071, IETF. `http://www.rfc-editor.org/rfc/rfc6071.txt`.

[97] Friedewald, M. and Raabe, O. (2011). Ubiquitous computing: An overview of technology impacts. *Telematics and Informatics*, 28(2):55–65.

[98] Gama, J., Medas, P., Castillo, G., and Rodrigues, P. (2004). Learning with drift detection. In *Advances in Artificial Intelligence–SBIA 2004: 17th Brazilian Symposium on Artificial Intelligence, Sao Luis, Maranhao, Brazil, September 29-Ocotber 1, 2004. Proceedings 17*, pages 286–295. Springer.

[99] Gama, J. and Pinto, C. (2006). Discretization from data streams: applications to histograms and data mining. In *Proc. of the 2006 ACM symposium on Applied computing*, pages 662–667. ACM.

[100] Gama, J., Sebastião, R., and Rodrigues, P. P. (2009). Issues in evaluation of stream learning algorithms. In *Proc. of the 15th ACM SIGKDD Int. Conf. on Knowledge discovery and data mining*, pages 329–338. ACM.

[101] Gama, J., Sebastião, R., and Rodrigues, P. P. (2013). On evaluating stream learning algorithms. *Machine learning*, 90(3):317–346.

[102] Gama, J., Žliobaitė, I., Bifet, A., Pechenizkiy, M., and Bouchachia, A. (2014). A survey on concept drift adaptation. *ACM computing surveys (CSUR)*, 46(4):1–37.

[103] Garcia, J., Hammad, M., and Malek, S. (2018). Lightweight, obfuscation-resilient detection and family identification of android malware. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 26(3):1–29.

[104] Garcia, S., Luengo, J., Saez, J. A., Lopez, V., and Herrera, F. (2012). A survey of discretization techniques: Taxonomy and empirical analysis in supervised learning. *IEEE Trans. on Knowledge and Data Engineering*, 25(4):734–750.

[105] Garcia-Teodoro, P., Diaz-Verdejo, J., Maciá-Fernández, G., and Vázquez, E. (2009). Anomaly-based network intrusion detection: Techniques, systems and challenges. *Computers & Security*, 28(1-2):18–28.

[106] Giacinto, G., Perdisci, R., Del Rio, M., and Roli, F. (2008). Intrusion detection in computer networks by a modular ensemble of one-class classifiers. *Inform. Fusion*, 9(1):69–82.

[107] Golovko, V. A., Vaitsekhovich, L. U., Kochurko, P. A., and Rubanau, U. S. (2007). Dimensionality reduction and attack recognition using neural network approaches. In *Neural Networks, 2007. IJCNN 2007. Int. Joint Conf. on*, pages 2734–2739. IEEE.

[108] Gonçalves Jr, P. M., de Carvalho Santos, S. G., Barros, R. S., and Vieira, D. C. (2014). A comparative study on concept drift detectors. *Expert Systems with Applications*, 41(18):8144–8156.

[109] Goodall, J. R., Ragan, E. D., Steed, C. A., Reed, J. W., Richardson, G. D., Huffer, K. M., Bridges, R. A., and Laska, J. A. (2018). Situ: Identifying and explaining suspicious behavior in networks. *IEEE transactions on visualization and computer graphics*, 25(1):204–214.

[110] Goodfellow, I., Bengio, Y., Courville, A., and Bengio, Y. (2016). *Deep learning*, volume 1. MIT press Cambridge.

[111] Google (2019a). Android security & privacy. 2018 year in review. `https://source.android.com/security/reports/Google_Android_Security_2018_Report_Final.pdf`.

[112] Google (2019b). Android security 2017 year in review. `https://source.android.com/security/reports/Google_Android_Security_2017_Report_Final.pdf`.

[113] Google (2020). Dalvik executable format. Accessed online: November 2, 2023.

[114] Görnitz, N., Kloft, M., Rieck, K., and Brefeld, U. (2009). Active learning for network intrusion detection. In *Proceedings of the 2nd ACM workshop on Security and artificial intelligence*, pages 47–54.

[115] Gorrie, M. (2022). What is a pua (potentially unwanted application) or pup (potentially unwanted program)? [Online] Available: `https://us.norton.com/internetsecurity-malware-what-are-puas-potentially-unwanted-applications.html`.

[116] Grace, M., Zhou, Y., Zhang, Q., Zou, S., and Jiang, X. (2012). Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 281–294.

[117] Guerra-Manzanares, A. and Bahsi, H. (2022). On the relativity of time: Implications and challenges of data drift on long-term effective android malware detection. *Computers & Security*, 122:102835.

[118] Guo, C., Ping, Y., Liu, N., and Luo, S.-S. (2016). A two-level hybrid approach for intrusion detection. *Neurocomputing*, 214:391–400.

[119] Guzella, T. S. and Caminhas, W. M. (2009). A review of machine learning approaches to spam filtering. *Expert Systems with Applications*, 36(7):10206–10222.

[120] Hahsler, M. and Dunham, M. H. (2011). Temporal structure learning for clustering massive data streams in real-time. In *Proc. of the 2011 SIAM Int. Conf. on Data Mining*, pages 664–675. SIAM.

[121] Haider, W., Hu, J., Slay, J., Turnbull, B. P., and Xie, Y. (2017). Generating realistic intrusion detection system dataset based on fuzzy qualitative modeling. *Journal of Network and Computer Applications*, 87:185–192.

[122] Hall, M. A. (1999). *Correlation-based Feature Selection for Machine Learning.* PhD thesis, The University of Waikato.

[123] Hamed, T., Dara, R., and Kremer, S. C. (2018). Network intrusion detection system based on recursive feature addition and bigram technique. *Computers & Security*, 73:137–155.

[124] Hammad, M., Garcia, J., and Malek, S. (2018). A large-scale empirical study on the effects of code obfuscations on android apps and anti-malware products. In *Proceedings of the 40th International Conference on Software Engineering*, pages 421–431.

[125] Handa, A., Sharma, A., and Shukla, S. K. (2019). Machine learning in cybersecurity: A review. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 9(4):e1306.

[126] Hansman, S. and Hunt, R. (2005). A taxonomy of network and computer attacks. *Computers & Security*, 24(1):31–43.

[127] Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., et al. (2020). Array programming with numpy. *Nature*, 585(7825):357–362.

[128] Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The elements of statistical learning: data mining, inference, and prediction.* Springer Science & Business Media.

[129] He, H. and Garcia, E. A. (2009). Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering*, 21(9):1263–1284.

[130] Heady, R., Luger, G. F., Maccabe, A., and Servilla, M. (1990). *The architecture of a network level intrusion detection system.* University of New Mexico.

[131] Hernández-González, J., Inza, I., and Lozano, J. A. (2016). Weak supervision and other non-standard classification problems: a taxonomy. *Pattern Recognition Letters*, 69:49–55.

[132] Hinkley, D. V. (1970). Inference about the change-point in a sequence of random variables. *Biometrika*, 57(1):1–17.

[133] Hodge, V. and Austin, J. (2004). A survey of outlier detection methodologies. *Artificial intelligence review*, 22(2):85–126.

[134] Hofstede, R., Čeleda, P., Trammell, B., Drago, I., Sadre, R., Sperotto, A., and Pras, A. (2014). Flow monitoring explained: From packet capture to data analysis with netflow and ipfix. *IEEE Communications Surveys & Tutorials*, 16(4):2037–2064.

[135] Homoliak, I. and Hanacek, P. (2019). Asnm datasets: A collection of network traffic features for testing of adversarial classifiers and network intrusion detectors. *arXiv preprint arXiv:1910.10528*.

[136] Homoliak, I., Teknøs, M., Ochoa, M., Breitenbacher, D., Hosseini, S., and Hanacek, P. (2018). Improving network intrusion detection classifiers by non-payload-based exploit-independent obfuscations: An adversarial approach. *EAI Endorsed Transactions on Security and Safety*, 5(17).

[137] Hoque, M. S., Mukit, M. A., and Bikas, M. A. N. (2012). An implementation of intrusion detection system using genetic algorithm. *Int. Journal of Network Security & Its Applications*, 4(2):109.

[138] Hoque, N., Bhuyan, M. H., Baishya, R. C., Bhattacharyya, D. K., and Kalita, J. K. (2014). Network attacks: Taxonomy, tools and systems. *Journal of Network and Computer Applications*, 40:307–324.

[139] Horchulhack, P., Viegas, E. K., and Santin, A. O. (2022). Toward feasible machine learning model updates in network-based intrusion detection. *Computer Networks*, 202:108618.

[140] Hu, W., Gao, J., Wang, Y., Wu, O., and Maybank, S. (2014). Online adaboost-based parameterized methods for dynamic distributed network intrusion detection. *IEEE Trans. on Cybernetics*, 44(1):66–82.

[141] Huang, T., Sethu, H., and Kandasamy, N. (2016). A new approach to dimensionality reduction for anomaly detection in data traffic. *IEEE Trans. on Network and Service Management*, 13(3):651–665.

[142] Hulten, G., Spencer, L., and Domingos, P. (2001). Mining time-changing data streams. In *ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, pages 97–106. ACM Press.

[143] Hyvärinen, A. and Oja, E. (2000). Independent component analysis: algorithms and applications. *Neural Networks*, 13(4-5):411–430.

[144] Jacobson, V., Leres, C., and McCanne, S. (1989). The tcpdump manual page. *Lawrence Berkeley Laboratory, Berkeley, CA*, 143.

[145] Jensen, A. and la Cour-Harbo, A. (2001). *Ripples in mathematics: the discrete wavelet transform*. Springer Science & Business Media.

[146] Jeon, S. and Kim, H. K. (2021). Autovas: An automated vulnerability analysis system with a deep learning approach. *Computers & Security*, 106:102308.

[147] Jerome, Q., Allix, K., State, R., and Engel, T. (2014). Using opcode-sequences to detect malicious android applications. In *2014 IEEE international conference on communications (ICC)*, pages 914–919. IEEE.

[148] Ji, S.-Y., Jeong, B.-K., Choi, S., and Jeong, D. H. (2016). A multi-level intrusion detection method for abnormal network behaviors. *Journal of Network and Computer Applications*, 62:9–17.

[149] Jin, S., Yeung, D. S., and Wang, X. (2007). Network intrusion detection in covariance feature space. *Pattern Recognition*, 40(8):2185–2197.

[150] Jolliffe, I. (2011). Principal component analysis. In Lovric, M., editor, *Int. Encyclopedia of Stat. Sci.*, pages 1094–1096. Springer Berlin Heidelberg, Berlin, Heidelberg.

[151] jun Huang, S., Jin, R., and Zhou, Z.-H. (2010). Active learning by querying informative and representative examples. In Lafferty, J. D., Williams, C. K. I., Shawe-Taylor, J., Zemel, R. S., and Culotta, A., editors, *Advances in Neural Information Processing Systems 23*, pages 892–900. Curran Associates, Inc.

[152] Kan, Z., Pendlebury, F., Pierazzi, F., and Cavallaro, L. (2021). Investigating labelless drift adaptation for malware detection. In *Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security*, pages 123–134.

[153] Kang, B., Yerima, S. Y., McLaughlin, K., and Sezer, S. (2016). N-opcode analysis for android malware classification and categorization. In *2016 International conference on cyber security and protection of digital services (cyber security)*, pages 1–7. IEEE.

[154] Kantchelian, A., Tschantz, M. C., Afroz, S., Miller, B., Shankar, V., Bachwani, R., Joseph, A. D., and Tygar, J. D. (2015). 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.

[155] Karbab, E. B. and Debbabi, M. (2021). Petadroid: Adaptive android malware detection using deep learning. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 319–340. Springer.

[156] Karlzen, H. and Sommestad, T. (2023). Automatic incident response solutions: a review of proposed solutions' input and output. In *Proceedings of the 18th International Conference on Availability, Reliability and Security*, pages 1–9.

[157] Kearns, M. J. (1990). *The Computational Complexity of Machine Learning.* PhD thesis, Harvard University, USA.

[158] Kendall, K. R. (1999). *A database of computer attacks for the evaluation of intrusion detection systems.* PhD thesis, Massachusetts Institute of Technology.

[159] Khan, L., Awad, M., and Thuraisingham, B. (2007). A new intrusion detection system using support vector machines and hierarchical clustering. *The VLDB journal*, 16(4):507–521.

[160] Khraisat, A., Gondal, I., Vamplew, P., and Kamruzzaman, J. (2019). Survey of intrusion detection systems: techniques, datasets and challenges. *Cybersecurity*, 2(1):20.

[161] Kim, D. S., Nguyen, H.-N., and Park, J. S. (2005). Genetic algorithm to improve svm based network intrusion detection system. In *Advanced Inform. Networking and Applications, 2005. AINA 2005. 19th Int. Conf. on*, volume 2, pages 155–158. IEEE.

[162] Kim, G., Lee, S., and Kim, S. (2014). A novel hybrid intrusion detection method integrating anomaly detection with misuse detection. *Expert Systems with Applications*, 41(4):1690–1700.

[163] Kim, J., Kim, J., Thu, H. L. T., and Kim, H. (2016). Long short term memory recurrent neural network classifier for intrusion detection. In *2016 Int. Conf. on Platform Technology and Service (PlatCon)*, pages 1–5. IEEE.

[164] Kim, T., Kang, B., Rho, M., Sezer, S., and Im, E. G. (2018). A multimodal deep learning method for android malware detection using various features. *IEEE Transactions on Information Forensics and Security*, 14(3):773–788.

[165] Kloft, M. and Laskov, P. (2010). Online anomaly detection under adversarial impact. In *Proc. of the 13th Int. Conf. on Artificial Intelligence and Statistics*, pages 405–412.

[166] Koc, L., Mazzuchi, T. A., and Sarkani, S. (2012). A network intrusion detection system based on a hidden naïve bayes multiclass classifier. *Expert Systems with Applications*, 39(18):13492–13500.

[167] Koch, R., Golling, M., and Rodosek, G. D. (2014). Behavior-based intrusion detection in encrypted environments. *IEEE Comm. Magazine*, 52(7):124–131.

[168] Kohonen, T. (2012). *Self-organization and associative memory*, volume 8. Springer Science & Business Media.

[169] Koli, J. (2018). Randroid: Android malware detection using random machine learning classifiers. In *2018 Technologies for Smart-City Energy Security and Power (ICSESP)*, pages 1–6. IEEE.

[170] Koufakou, A. and Georgiopoulos, M. (2010). A fast outlier detection strategy for distributed high-dimensional data sets with mixed attributes. *Data Mining and Knowledge Discovery*, 20(2):259–289.

[171] Kovanen, T., David, G., and Hämäläinen, T. (2016). Survey: Intrusion detection systems in encrypted traffic. In *Internet of Things, Smart Spaces, and Next Generation Networks and Systems*, pages 281–293. Springer.

[172] Krempl, G., Žliobaite, I., Brzeziundefinedski, D., Hüllermeier, E., Last, M., Lemaire, V., Noack, T., Shaker, A., Sievi, S., Spiliopoulou, M., and Stefanowski, J. (2014). Open challenges for data stream mining research. *SIGKDD Explor. Newsl.*, 16(1):1–10.

[173] Krüegel, C., Toth, T., and Kirda, E. (2002). Service specific anomaly detection for network intrusion detection. In *Proc. of the 2002 ACM Symposium on Applied Computing*, SAC '02, pages 201–208, New York, NY, USA. ACM.

[174] Labs, K. (2021). Mobile malware evolution 2020. [Online] Available: `https://securelist.com/mobile-malware-evolution-2020/101029/`.

[175] Lakhina, A., Crovella, M., and Diot, C. (2005). Mining anomalies using traffic feature distributions. *ACM SIGCOMM computer communication review*, 35(4):217–228.

[176] Lane, T. and Brodley, C. E. (1999). Temporal sequence learning and data reduction for anomaly detection. *ACM Trans. on Inform. and System Security (TISSEC)*, 2(3):295–331.

[177] Lashkari, A. H., Kadir, A. F. A., Taheri, L., and Ghorbani, A. A. (2018). Toward developing a systematic approach to generate benchmark android malware datasets and classification. In *2018 International Carnahan Conference on Security Technology (ICCST)*, pages 1–7. IEEE.

[178] LeCun, Y., Bottou, L., Orr, G. B., and Müller, K.-R. (1998). Efficient backprop. In *Neural Networks: Tricks of the Trade*, pages 9–50. Springer.

[179] Lee, W. and Stolfo, S. J. (2000). A framework for constructing features and models for intrusion detection systems. *ACM Trans. on Inform. and System Security (TiSSEC)*, 3(4):227–261.

[180] Lee, W. Y., Saxe, J., and Harang, R. (2019). Seqdroid: Obfuscated android malware detection using stacked convolutional and recurrent neural networks. In *Deep learning applications for cyber security*, pages 197–210. Springer.

[181] Lei, J. Z. and Ghorbani, A. A. (2012). Improved competitive learning neural networks for network intrusion and fraud detection. *Neurocomputing*, 75(1):135–145.

[182] Li, J., Chen, R., Su, J., Huang, X., and Wang, X. (2019). Me-tls: Middlebox-enhanced tls for internet-of-things devices. *IEEE Internet of Things Journal*.

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

[184] Li, Y., Jang, J., Hu, X., and Ou, X. (2017b). Android malware clustering through malicious payload mining. In *International symposium on research in attacks, intrusions, and defenses*, pages 192–214. Springer.

[185] Lin, S.-W., Ying, K.-C., Lee, C.-Y., and Lee, Z.-J. (2012). An intelligent algorithm with feature selection and decision rules applied to anomaly intrusion detection. *Applied Soft Computing*, 12(10):3285–3290.

[186] Lin, W.-C., Ke, S.-W., and Tsai, C.-F. (2015). Cann: An intrusion detection system based on combining cluster centers and nearest neighbors. *Knowledge-Based Systems*, 78:13–21.

[187] Lindorfer, M., Volanis, S., Sisto, A., Neugschwandtner, M., Athanasopoulos, E., Maggi, F., Platzer, C., Zanero, S., and Ioannidis, S. (2014). Andradar: fast discovery of android applications in alternative markets. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 51–71. Springer.

[188] Lippmann, R., Haines, J. W., Fried, D. J., Korba, J., and Das, K. (2000). The 1999 darpa off-line intrusion detection evaluation. *Comput. Netw.*, 34(4):579–595.

[189] Liu, H., Hussain, F., Tan, C. L., and Dash, M. (2002). Discretization: An enabling technique. *Data mining and Knowledge Discovery*, 6(4):393–423.

[190] Liu, K., Xu, S., Xu, G., Zhang, M., Sun, D., and Liu, H. (2020). A review of android malware detection approaches based on machine learning. *IEEE Access*, 8:124579–124607.

[191] LLC, Q. (2019). Argus: Auditing network activity. `https://qosient.com/argus/index.shtml`.

[192] Lopez-Martin, M., Carro, B., and Sanchez-Esguevillas, A. (2020). Application of deep reinforcement learning to intrusion detection for supervised problems. *Expert Systems with Applications*, 141:112963.

[193] Lu, J., Liu, A., Dong, F., Gu, F., Gama, J., and Zhang, G. (2019). Learning under concept drift: A review. *IEEE Transactions on Knowledge and Data Engineering*, 31(12):2346–2363.

[194] Macario, G., Torchiano, M., and Violante, M. (2009). An in-vehicle infotainment software architecture based on google android. In *2009 IEEE International Symposium on Industrial Embedded Systems*, pages 257–260. IEEE.

[195] Maciá-Fernández, G., Camacho, J., Magán-Carrión, R., García-Teodoro, P., and Therón, R. (2018). Ugr '16: A new dataset for the evaluation of cyclostationarity-based network idss. *Computers & Security*, 73:411–424.

[196] Mahindru, A. and Sangal, A. (2021). Semidroid: a behavioral malware detector based on unsupervised machine learning techniques using feature selec-

tion approaches. *International Journal of Machine Learning and Cybernetics*, 12(5):1369–1411.

[197] Mahoney, M. V. and Chan, P. K. (2003a). An analysis of the 1999 darpa/lincoln laboratory evaluation data for network anomaly detection. In *Int. Wksp. on Recent Advances in Intrusion Detection*, pages 220–237. Springer.

[198] Mahoney, M. V. and Chan, P. K. (2003b). Learning rules for anomaly detection of hostile network traffic. In *Third IEEE Int. Conf. on Data Mining*, pages 601–604. IEEE.

[199] Mai, J., Chuah, C.-N., Sridharan, A., Ye, T., and Zang, H. (2006). Is sampled data sufficient for anomaly detection? In *Proc. of the 6th ACM SIGCOMM Conf. on Internet measurement*, pages 165–176. ACM.

[200] Maimon, O. and Rokach, L. (2009). Introduction to knowledge discovery and data mining. In *Data mining and knowledge discovery handbook*, pages 1–15. Springer.

[201] Maiorca, D., Ariu, D., Corona, I., Aresu, M., and Giacinto, G. (2015). Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security*, 51:16–31.

[202] Maiwald, E. (2001). *Network security: a beginner's guide*. McGraw-Hill Professional.

[203] McHugh, J. (2000). Testing intrusion detection systems: a critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory. *ACM Trans. on Inform. and System Security (TISSEC)*, 3(4):262–294.

[204] McLaughlin, N., Martinez del Rincon, J., Kang, B., Yerima, S., Miller, P., Sezer, S., Safaei, Y., Trickel, E., Zhao, Z., Doupé, A., et al. (2017). Deep android malware detection. In *Proceedings of the seventh ACM on conference on data and application security and privacy*, pages 301–308.

[205] Miller, B., Kantchelian, A., Tschantz, M. C., Afroz, S., Bachwani, R., Faizullabhoy, R., Huang, L., Shankar, V., Wu, T., Yiu, G., et al. (2016). Reviewer integration and performance measurement for malware detection. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 122–141. Springer.

[206] Mivule, K. and Anderson, B. (2015). A study of usability-aware network trace anonymization. In *2015 Science and Information Conference (SAI)*, pages 1293–1304. IEEE.

[207] Molina-Coronado, B., Mori, U., Mendiburu, A., and Miguel-Alonso, J. (2020). Survey of network intrusion detection methods from the perspective of the knowledge discovery in databases process. *IEEE Transactions on Network and Service Management*, 17(4):2451–2479.

[208] Molina-Coronado, B., Mori, U., Mendiburu, A., and Miguel-Alonso, J. (2023a). Efficient concept drift handling for batch android malware detection models.

[209] Molina-Coronado, B., Mori, U., Mendiburu, A., and Miguel-Alonso, J. (2023b). Towards a fair comparison and realistic evaluation framework of android

malware detectors based on static analysis and machine learning. *Computers & Security*, 124:102996.

[210] Moser, A., Kruegel, C., and Kirda, E. (2007). Limits of static analysis for malware detection. In *Twenty-third annual computer security applications conference (ACSAC 2007)*, pages 421–430. IEEE.

[211] Moustafa, N., Hu, J., and Slay, J. (2019). A holistic review of network anomaly detection systems: A comprehensive survey. *Journal of Network and Computer Applications*, 128:33–55.

[212] Moustafa, N. and Slay, J. (2015). Unsw-nb15: a comprehensive data set for network intrusion detection systems (unsw-nb15 network data set). In *Military Comm. and Inform. Systems Conf. (MilCIS), 2015*, pages 1–6. IEEE.

[213] Mukkamala, S., Janoski, G., and Sung, A. (2002). Intrusion detection using neural networks and support vector machines. In *Neural Networks, 2002. IJCNN'02. Proc. of the 2002 Int. Joint Conf. on*, volume 2, pages 1702–1707. IEEE.

[214] Murdoch, S. J. and Danezis, G. (2005). Low-cost traffic analysis of tor. In *2005 IEEE Symposium on Security and Privacy (S&P'05)*, pages 183–195. IEEE.

[215] Narayanan, A., Chandramohan, M., Chen, L., and Liu, Y. (2017). Context-aware, adaptive, and scalable android malware detection through online learning. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 1(3):157–175.

[216] Nenkova, A. and McKeown, K. (2012). *A survey of text summarization techniques*, pages 43–76. Springer, Boston, MA.

[217] Newsome, J., Karp, B., and Song, D. (2006). Paragraph: Thwarting signature learning by training maliciously. In *Int. Wksp. on Recent Advances in Intrusion Detection*, pages 81–105. Springer.

[218] Nguyen, H. H., Harbi, N., and Darmont, J. (2011). An efficient local region and clustering-based ensemble system for intrusion detection. In *Proc. of the 15th Symposium on Int. Database Engineering & Applications*, pages 185–191. ACM.

[219] Nguyen, H.-L., Woon, Y.-K., and Ng, W.-K. (2015). A survey on data stream clustering and classification. *Knowledge and Inform. Systems*, 45(3):535–569.

[220] Nicolau, M., McDermott, J., et al. (2018). Learning neural representations for network anomaly detection. *IEEE Trans. on Cybernetics*, 49(8):3074–3087.

[221] Oltsik, J. (2015). White paper: Network encryption and its impact on network security. Technical report, Enterprise Strategy Group, Inc. accessed February 2019.

[222] Onwuzurike, L., Mariconti, E., Andriotis, P., Cristofaro, E. D., Ross, G., and Stringhini, G. (2019). Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version). *ACM Transactions on Privacy and Security (TOPS)*, 22(2):1–34.

[223] Otey, M. E., Ghoting, A., and Parthasarathy, S. (2006). Fast distributed outlier detection in mixed-attribute data sets. *Data Mining and Knowledge Discovery*, 12(2-3):203–228.

[224] Page, E. S. (1954). Continuous inspection schemes. *Biometrika*, 41(1/2):100–115.

[225] Palmieri, F., Fiore, U., and Castiglione, A. (2014). A distributed approach to network anomaly detection based on independent component analysis. *Concurrency and Computation: Practice and Experience*, 26(5):1113–1129.

[226] Pan, Y., Ge, X., Fang, C., and Fan, Y. (2020). A systematic literature review of android malware detection using static analysis. *IEEE Access*, 8:116363–116379.

[227] Papamartzivanos, D., Mármol, F. G., and Kambourakis, G. (2019). Introducing deep learning self-adaptive misuse network intrusion detection systems. *IEEE Access*, 7:13546–13560.

[228] Papernot, N., McDaniel, P., Goodfellow, I., Jha, S., Celik, Z. B., and Swami, A. (2017). Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, pages 506–519.

[229] Paxson, V. (1999). Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463.

[230] Peddabachigari, S., Abraham, A., Grosan, C., and Thomas, J. (2007). Modeling intrusion detection system using hybrid intelligent systems. *Journal of Network and Computer Applications*, 30(1):114–132.

[231] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

[232] Pendlebury, F., Pierazzi, F., Jordaney, R., Kinder, J., and Cavallaro, L. (2019). {TESSERACT}: Eliminating experimental bias in malware classification across space and time. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 729–746.

[233] Perdisci, R., Ariu, D., Fogla, P., Giacinto, G., and Lee, W. (2009). Mcpad: A multiple classifier system for accurate payload-based anomaly detection. *Computer Networks*, 53(6):864–881.

[234] Piateski, G. and Frawley, W. (1991). *Knowledge discovery in databases*. MIT press.

[235] Pimentel, M. A., Clifton, D. A., Clifton, L., and Tarassenko, L. (2014). A review of novelty detection. *Signal Processing*, 99:215–249.

[236] Portnoy, L., Eskin, E., and Stolfo, S. (2001). Intrusion detection with unlabeled data using clustering. In *In Proc. of ACM CSS Wksp. on Data Mining Applied to Security (DMSA-2001)*. ACM.

[237] Preda, M. D. and Maggi, F. (2017). Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology. *Journal of Computer Virology and Hacking Techniques*, 13(3):209–232.

[238] Ptacek, T. H. and Newsham, T. N. (1998). Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks inc Calgary Alberta.

[239] Quinlan, J. R. (1993). *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[240] Quinlan, J. R. (1996a). Boosting, bagging, and c4. 5. In *Proc. of the 13th National Conf. on Artificial Intelligence*, pages 725–730. AAAI Press Menlo Park, CA.

[241] Quinlan, J. R. (1996b). Improved use of continuous attributes in c4.5. *Journal of Artificial Intelligence Research*, 4:77–90.

[242] Quittek, J., Zseby, T., Claise, B., and Zander, S. (2004). Requirements for ip flow information export (ipfix). RFC 3917, IETF.

[243] Raina, R., Battle, A., Lee, H., Packer, B., and Ng, A. Y. (2007). Self-taught learning: transfer learning from unlabeled data. In *Proc. of the 24th Int. Conf. on Machine learning*, pages 759–766. ACM.

[244] Ramírez-Gallego, S., García, S., and Herrera, F. (2018). Online entropy-based discretization for data streaming classification. *Future Generation Computer Systems*, 86:59–70.

[245] Raspall, F. (2012). Efficient packet sampling for accurate traffic measurements. *Computer Networks*, 56(6):1667–1684.

[246] Rastogi, V., Chen, Y., and Jiang, X. (2013). Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 329–334.

[247] Rastogi, V., Chen, Y., and Jiang, X. (2014). Catch me if you can: Evaluating android anti-malware against transformation attacks. *IEEE Transactions on Information Forensics and Security*, 9(1):99–108.

[248] Ratabouil, S. (2015). *Android NDK: beginner's guide*. Packt Publishing Ltd.

[249] Raudys, S. J., Jain, A. K., et al. (1991). Small sample size effects in statistical pattern recognition: Recommendations for practitioners. *IEEE Transactions on pattern analysis and machine intelligence*, 13(3):252–264.

[250] Roshan, S., Miche, Y., Akusok, A., and Lendasse, A. (2018). Adaptive and online network intrusion detection system using clustering and extreme learning machines. *Journal of the Franklin Institute*, 355(4):1752–1779.

[251] Roy, S., DeLoach, J., Li, Y., Herndon, N., Caragea, D., Ou, X., Ranganath, V. P., Li, H., and Guevara, N. (2015). Experimental study with real-world data for android app security analysis using machine learning. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 81–90.

[252] Rubin, J. B., Besehanic, J., and Borland, R. P. (2014). Intercepting encrypted network traffic for internet usage monitoring. US Patent 8,914,629.

[253] Ruggia, A., Losiouk, E., Verderame, L., Conti, M., and Merlo, A. (2021). Repack me if you can: An anti-repackaging solution based on android virtualization. In *Annual Computer Security Applications Conference*, pages 970–981.

[254] Saeys, Y., Inza, I., and Larrañaga, P. (2007). A review of feature selection techniques in bioinformatics. *Bioinformatics*, 23(19):2507–2517.

[255] Salem, A., Banescu, S., and Pretschner, A. (2019). Don't pick the cherry: An evaluation methodology for android malware detection methods. *arXiv preprint arXiv:1903.10560*.

[256] Salem, A., Banescu, S., and Pretschner, A. (2021). Maat: Automatically analyzing virustotal for accurate labeling and effective malware detection. *ACM Transactions on Privacy and Security (TOPS)*, 24(4):1–35.

[257] Salo, F., Nassif, A. B., and Essex, A. (2019). Dimensionality reduction with ig-pca and ensemble classifier for network intrusion detection. *Computer Networks*, 148:164–175.

[258] Santafe, G., Inza, I., and Lozano, J. A. (2015). Dealing with the evaluation of supervised classification algorithms. *Artificial Intelligence Review*, 44(4):467–508.

[259] Sarasamma, S. T., Zhu, Q. A., and Huff, J. (2005). Hierarchical kohonenen net for anomaly detection in network security. *IEEE Trans. on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 35(2):302–312.

[260] Satten, C. (2008). Lossless gigabit remote packet capture with linux. `https://staff.washington.edu/corey/gulp/`.

[261] Sebastián, S. and Caballero, J. (2020). Avclass2: Massive malware tag extraction from av labels. In *Annual Computer Security Applications Conference*, pages 42–53.

[262] Selvakumar, B. and Muneeswaran, K. (2019). Firefly algorithm based feature selection for network intrusion detection. *Computers & Security*, 81:148–155.

[263] Seni, G. and Elder, J. F. (2010). Ensemble methods in data mining: improving accuracy through combining predictions. *Synthesis lectures on data mining and knowledge discovery*, 2(1):1–126.

[264] Sheikhan, M., Jadidi, Z., and Farrokhi, A. (2012). Intrusion detection using reduced-size rnn based on feature grouping. *Neural Computing and Applications*, 21(6):1185–1190.

[265] Shiravi, A., Shiravi, H., Tavallaee, M., and Ghorbani, A. A. (2012). Toward developing a systematic approach to generate benchmark datasets for intrusion detection. *Computers & Security*, 31(3):357–374.

[266] Shone, N., Ngoc, T. N., Phai, V. D., and Shi, Q. (2018). A deep learning approach to network intrusion detection. *IEEE Trans. on Emerging Topics in Computational Intelligence*, 2(1):41–50.

[267] Shyu, M.-L., Chen, S.-C., Sarinnapakorn, K., and Chang, L. (2003). A novel anomaly detection scheme based on principal component classifier. *Proc. of Int. Conf. on Data Mining.*

[268] Siddique, K., Akhtar, Z., Khan, F. A., and Kim, Y. (2019). Kdd cup 99 data sets: A perspective on the role of data sets in network intrusion detection research. *Computer*, 52(2):41–51.

[269] Sihag, V., Vardhan, M., and Singh, P. (2021). A survey of android application and malware hardening. *Computer Science Review*, 39:100365.

[270] Silva, J. A., Faria, E. R., Barros, R. C., Hruschka, E. R., De Carvalho, A. C., and Gama, J. (2013). Data stream clustering: A survey. *ACM Computing Surveys (CSUR)*, 46(1):13.

[271] Sommer, R. and Paxson, V. (2010). Outside the closed world: On using machine learning for network intrusion detection. In *2010 IEEE Symp. on Security and Privacy*, pages 305–316.

[272] Song, J., Takakura, H., Okabe, Y., Eto, M., Inoue, D., and Nakao, K. (2011). Statistical analysis of honeypot data and building of kyoto 2006+ dataset for nids evaluation. In *Proc. of the 1st Wksp. on Building Analysis Datasets and Gathering Experience Returns for Security*, pages 29–36. ACM.

[273] Souri, A. and Hosseini, R. (2018). A state-of-the-art survey of malware detection approaches using data mining techniques. *Human-centric Computing and Information Sciences*, 8(1):3.

[274] Sperotto, A., Schaffrath, G., Sadre, R., Morariu, C., Pras, A., and Stiller, B. (2010). An overview of ip flow-based intrusion detection. *IEEE Comm. Surveys & Tutorials*, 12(3):343–356.

[275] Spreitzenbarth, M., Schreck, T., Echtler, F., Arp, D., and Hoffmann, J. (2015). Mobile-sandbox: combining static and dynamic analysis with machine-learning techniques. *International Journal of Information Security*, 14(2):141–153.

[276] Stallings, W. (2004). *Computer networking with Internet protocols and technology*. Pearson/Prentice Hall Upper Saddle River, NJ, USA.

[277] Statista (2022). Mobile operating systems' market share worldwide from january 2012 to january 2022. [Online] Available: `https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/`.

[278] Stytz, M. R. and Banks, S. B. (2005). Method and apparatus for preventing network traffic analysis. US Patent 6,917,974.

[279] Suarez-Tangil, G., Dash, S. K., Ahmadi, M., Kinder, J., Giacinto, G., and Cavallaro, L. (2017). 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.

[280] Surendran, R. (2021). On impact of semantically similar apps in android malware datasets. *ArXiv*, abs/2112.02606.

[281] Tam, K., Feizollah, A., Anuar, N. B., Salleh, R., and Cavallaro, L. (2017). The evolution of android malware and android analysis techniques. *ACM Computing Surveys (CSUR)*, 49(4):1–41.

[282] Tan, D. J., Chua, T.-W., and Thing, V. L. (2015). Securing android: a survey, taxonomy, and challenges. *ACM Computing Surveys (CSUR)*, 47(4):1–45.

[283] Tan, Z., Jamdagni, A., He, X., Nanda, P., and Liu, R. P. (2013). A system for denial-of-service attack detection based on multivariate correlation analysis. *IEEE Trans. on Parallel and Distributed Systems*, 25(2):447–456.

[284] Tavallaee, M., Bagheri, E., Lu, W., and Ghorbani, A. A. (2009). A detailed analysis of the kdd cup 99 data set. In *Computational Intelligence for Security and Defense Applications, 2009. CISDA 2009. IEEE Symposium on*, pages 1–6. IEEE.

[285] Tax, D. and Duin, R. (2000). Feature scaling in support vector data descriptions. *Learning from Imbalanced Datasets*, pages 25–30.

[286] Tax, D. M. J. (2001). *One-class classification. Concept-learning in the absence of counter-examples*. PhD thesis, Delft University of Technology.

[287] Thottan, M. and Ji, C. (2003). Anomaly detection in ip networks. *IEEE Trans. on Signal Processing*, 51(8):2191–2204.

[288] Tsai, C.-F. and Lin, C.-Y. (2010). A triangle area based nearest neighbors approach to intrusion detection. *Pattern Recognition*, 43(1):222–229.

[289] Ucci, D., Aniello, L., and Baldoni, R. (2019). Survey of machine learning techniques for malware analysis. *Computers & Security*, 81:123–147.

[290] Umer, M. F., Sher, M., and Bi, Y. (2017). Flow-based intrusion detection: techniques and challenges. *Computers & Security*, 70:238–254.

[291] Van Engelen, J. E. and Hoos, H. H. (2020). A survey on semi-supervised learning. *Machine Learning*, 109(2):373–440.

[292] Viegas, E., Santin, A., Bessani, A., and Neves, N. (2019). Bigflow: Real-time and reliable anomaly-based intrusion detection for high-speed networks. *Future Generation Computer Systems*, 93:473–485.

[293] Viegas, E., Santin, A., Neves, N., Bessani, A., and Abreu, V. (2017a). A resilient stream learning intrusion detection mechanism for real-time analysis of network traffic. In *GLOBECOM 2017-2017 IEEE Global Comm. Conf.*, pages 1–6. IEEE.

[294] Viegas, E. K., Santin, A. O., and Oliveira, L. S. (2017b). Toward a reliable anomaly-based intrusion detection in real-world environments. *Computer Networks*, 127:200–216.

[295] Wang, W., Wang, X., Feng, D., Liu, J., Han, Z., and Zhang, X. (2014). Exploring permission-induced risk in android applications for malicious application detection. *IEEE Transactions on Information Forensics and Security*, 9(11):1869–1882.

[296] Wang, W., Zhao, M., Gao, Z., Xu, G., Xian, H., Li, Y., and Zhang, X. (2019). Constructing features for detecting android malicious applications: issues, taxonomy and directions. *IEEE access*, 7:67602–67631.

[297] Wang, X., Wang, W., He, Y., Liu, J., Han, Z., and Zhang, X. (2017). Characterizing android apps' behavior for effective detection of malapps at large scale. *Future generation computer systems*, 75:30–45.

[298] Webb, G. I., Hyde, R., Cao, H., Nguyen, H. L., and Petitjean, F. (2016). Characterizing concept drift. *Data Mining and Knowledge Discovery*, 30(4):964–994.

[299] Wei, F., Li, Y., Roy, S., Ou, X., and Zhou, W. (2017). Deep ground truth analysis of current android malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'17)*, pages 252–276, Bonn, Germany. Springer.

[300] Wei, Y., Bo, L., Wu, X., Li, Y., Ye, Z., Sun, X., and Li, B. (2023). Vulrep: vulnerability repair based on inducing commits and fixing commits. *EURASIP Journal on Wireless Communications and Networking*, 2023(1):1–16.

[301] Weller-Fahy, D. J., Borghetti, B. J., and Sodemann, A. A. (2014). A survey of distance and similarity measures used within network intrusion anomaly detection. *IEEE Comm. Surveys & Tutorials*, 17(1):70–91.

[302] Wheelus, C., Bou-Harb, E., and Zhu, X. (2018). Tackling class imbalance in cyber security datasets. In *2018 IEEE Int. Conf. on Inform. Reuse and Integration (IRI)*, pages 229–232. IEEE.

[303] Whitehill, J., Wu, T.-f., Bergsma, J., Movellan, J., and Ruvolo, P. (2009). Whose vote should count more: Optimal integration of labels from labelers of unknown expertise. *Advances in neural information processing systems*, 22:2035–2043.

[304] Wiesler, S. and Ney, H. (2011). A convergence analysis of log-linear training. In Shawe-Taylor, J., Zemel, R. S., Bartlett, P. L., Pereira, F., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 24*, pages 657–665. Curran Associates, Inc.

[305] Winter, P., Lampesberger, H., Zeilinger, M., and Hermann, E. (2011). On detecting abrupt changes in network entropy time series. In *IFIP Int. Conf. on Comm. and Multimedia Security*, pages 194–205. Springer.

[306] Wright, C. V., Coull, S. E., and Monrose, F. (2009). Traffic morphing: An efficient defense against statistical traffic analysis. In *NDSS*, volume 9. Internet Society.

[307] Wu, D.-J., Mao, C.-H., Wei, T.-E., Lee, H.-M., and Wu, K.-P. (2012). Droid-mat: Android malware detection through manifest and api calls tracing. In *2012 Seventh Asia Joint Conference on Information Security*, pages 62–69. IEEE.

[308] Wu, J. and Kanai, A. (2021). Utilizing obfuscation information in deep learning-based android malware detection. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1321–1326. IEEE.

[309] Xu, K., Li, Y., Deng, R., Chen, K., and Xu, J. (2019). Droidevolver: Self-evolving android malware detection system. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 47–62. IEEE.

[310] Xu, K., Li, Y., and Deng, R. H. (2016a). Iccdetector: Icc-based malware detection on android. *IEEE Transactions on Information Forensics and Security*, 11(6):1252–1264.

[311] Xu, M., Song, C., Ji, Y., Shih, M.-W., Lu, K., Zheng, C., Duan, R., Jang, Y., Lee, B., Qian, C., et al. (2016b). Toward engineering a secure android ecosystem: A survey of existing techniques. *ACM Computing Surveys (CSUR)*, 49(2):1–47.

[312] Yamada, A., Miyake, Y., Takemori, K., Studer, A., and Perrig, A. (2007). Intrusion detection for encrypted web accesses. In *21st Int. Conf. on Advanced Inform. Networking and Applications Wksp. (AINAW'07)*, volume 1, pages 569–576. IEEE.

[313] Yang, L., Ciptadi, A., Laziuk, I., Ahmadzadeh, A., and Wang, G. (2021a). Bodmas: An open dataset for learning based temporal analysis of pe malware. In *2021 IEEE Security and Privacy Workshops (SPW)*, pages 78–84. IEEE.

[314] Yang, L., Guo, W., Hao, Q., Ciptadi, A., Ahmadzadeh, A., Xing, X., and Wang, G. (2021b). {CADE}: Detecting and explaining concept drift samples for security applications. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2327–2344.

[315] Yang, Y. and Webb, G. I. (2001). Proportional k-interval discretization for naive-bayes classifiers. In *European Conf. on Machine Learning*, pages 564–575. Springer.

[316] Yang, Y. and Webb, G. I. (2002). A comparative study of discretization methods for naive-bayes classifiers. In *Proceedings of PKAW*, volume 2002.

[317] Ye, Y., Li, T., Adjeroh, D., and Iyengar, S. S. (2017). A survey on malware detection using data mining techniques. *ACM Computing Surveys (CSUR)*, 50(3):1–40.

[318] Yerima, S. Y., Sezer, S., McWilliams, G., and Muttik, I. (2013). A new android malware detection approach using bayesian classification. In *2013 IEEE 27th international conference on advanced information networking and applications (AINA)*, pages 121–128. IEEE.

[319] Yeung, D. S., Jin, S., and Wang, X. (2007). Covariance-matrix modeling and detecting various flooding attacks. *IEEE Trans. on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 37(2):157–169.

[320] Yin, C., Zhu, Y., Fei, J., and He, X. (2017). A deep learning approach for intrusion detection using recurrent neural networks. *IEEE Access*, 5:21954–21961.

[321] Yokoyama, A., Ishii, K., Tanabe, R., Papa, Y., Yoshioka, K., Matsumoto, T., Kasama, T., Inoue, D., Brengel, M., Backes, M., et al. (2016). Sandprint: Fingerprinting malware sandboxes to provide intelligence for sandbox evasion. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 165–187. Springer.

[322] You, I. and Yim, K. (2010). Malware obfuscation techniques: A brief survey. In *2010 International conference on broadband, wireless computing, communication and applications*, pages 297–300. IEEE.

[323] Yuan, Z., Lu, Y., and Xue, Y. (2016). Droiddetector: android malware characterization and detection using deep learning. *Tsinghua Science and Technology*, 21(1):114–123.

[324] Zaki, M. J. and Wagner Meira, J. (2014). *Data Mining and Analysis: Fundamental Concepts and Algorithms.* Cambridge University Press.

[325] Zhang, J. and Li, C. (2019). Adversarial examples: Opportunities and challenges. *IEEE transactions on neural networks and learning systems*, 31(7):2578–2593.

[326] Zhang, J. and Zulkernine, M. (2006). Anomaly based network intrusion detection with unsupervised outlier detection. In *Comm., 2006. ICC'06. IEEE Int. Conf. on*, volume 5, pages 2388–2393. IEEE.

[327] Zhang, J., Zulkernine, M., and Haque, A. (2008). Random-forests-based network intrusion detection systems. *IEEE Trans. on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 38(5):649–659.

[328] Zhang, X., Breitinger, F., Luechinger, E., and O'Shaughnessy, S. (2021). Android application forensics: A survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations. *Forensic Science International: Digital Investigation*, 39:301285.

[329] Zhang, X. and Jin, Z. (2016). A new semantics-based android malware detection. In *2016 2nd IEEE International Conference on Computer and Communications (ICCC)*, pages 1412–1416. IEEE.

[330] Zhang, X., Zhang, Y., Zhong, M., Ding, D., Cao, Y., Zhang, Y., Zhang, M., and Yang, M. (2020). Enhancing state-of-the-art classifiers with api semantics

to detect evolved android malware. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, pages 757–770.

[331] Zhao, Y., Li, L., Wang, H., Cai, H., Bissyandé, T. F., Klein, J., and Grundy, J. (2021). On the impact of sample duplication in machine-learning-based android malware detection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(3):1–38.

[332] Zhao, Z. and Liu, H. (2007). Searching for interacting features. In *Proc. of the 20th Int. Joint Conf. on Artifical Intelligence*, IJCAI'07, pages 1156–1161, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[333] Zhou, Y. and Jiang, X. (2012). Dissecting android malware: Characterization and evolution. In *2012 IEEE symposium on security and privacy*, pages 95–109. IEEE.

[334] Zhu, H.-J., You, Z.-H., Zhu, Z.-X., Shi, W.-L., Chen, X., and Cheng, L. (2018). Droiddet: effective and robust detection of android malware using static analysis along with rotation forest model. *Neurocomputing*, 272:638–646.

[335] Zhu, S., Shi, J., Yang, L., Qin, B., Zhang, Z., Song, L., and Wang, G. (2020). Measuring and modeling the label dynamics of online anti-malware engines. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2361–2378. USENIX Association.

[336] Zhu, X. and Wu, X. (2004). Class noise vs. attribute noise: A quantitative study. *Artificial Intelligence Review*, 22(3):177–210.

[337] Zliobaite, I. and Gabrys, B. (2012). Adaptive preprocessing for streaming data. *IEEE transactions on knowledge and data Engineering*, 26(2):309–321.

[338] Žliobaitė, I., Pechenizkiy, M., and Gama, J. (2016). An overview of concept drift applications. *Big data analysis: new algorithms for a new society*, pages 91–114.

eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea

Intelligent Systems Group