Università degli Studi di Cagliari

PHD DEGREE

Electronic and Computer Engineering
Cycle XXXIII

TITLE OF THE PHD THESIS

Malware Analysis and Detection with
Explainable Machine Learning

Scientific Disciplinary Sector
S.S.D. ING-INF/05

PhD Student: Michele Scalas
Supervisor: Prof. Giorgio Giacinto

Final Exam. Academic Year 2019-2020
Thesis defence: March 2021 Session

Malware Analysis and Detection with Explainable Machine Learning

Michele Scalas

Abstract

Malware detection is one of the areas where machine learning is successfully employed due to its high discriminating power and the capability of identifying novel variants of malware samples. Typically, the problem formulation is strictly correlated to the use of a wide variety of features covering several characteristics of the entities to classify. Apparently, this practice allows achieving considerable detection performance. However, it hardly permits us to gain insights into the knowledge extracted by the learning algorithm, causing two main issues. First, detectors might learn spurious patterns; thus, undermining their *effectiveness* in real environments. Second, they might be particularly vulnerable to adversarial attacks; thus, weakening their *security*. These concerns give rise to the necessity to develop systems tailored to the specific peculiarities of the attacks to detect.

Within malware detection, Android ransomware represents a challenging yet illustrative domain for assessing the relevance of this issue. Ransomware represents a serious threat that acts by locking the compromised device or encrypting its data, then forcing the device owner to pay a ransom in order to restore the device functionality. Attackers typically develop such dangerous apps so that normally-legitimate components and functionalities perform malicious behaviour; thus, making them harder to be distinguished from genuine applications. In this sense, adopting a well-defined variety of features and relying on some kind of explanations about the logic behind such detectors could improve their design process since it could reveal truly characterising features; hence, guiding the human expert towards the understanding of the most relevant attack patterns.

Given this context, the goal of the thesis is to explore strategies that may improve the design process of malware detectors. In particular, the thesis proposes to evaluate and integrate approaches based on rising research on *explainable machine learning*. To this end, the work follows two pathways. The first and main one focuses on identifying the main traits that result to be characterising and effective for Android ransomware detection. Then, explainability techniques are used to propose methods to assess the validity of the considered features. The second pathway broadens the view by exploring the relationship between explainable machine learning and adversarial attacks. In this regard, the contribution consists of pointing out metrics extracted from explainability techniques that can reveal models' robustness to adversarial attacks, together with an assessment of the practical feasibility for attackers to alter the features that affect models' output the most.

Ultimately, this work highlights the necessity to adopt a design process that is aware of the weaknesses and attacks against machine learning-based detectors, and proposes explainability techniques as one of the tools to counteract them.

1 There is an appointed time for
everything, and a time for every affair
under the heavens.
2 A time to be born, and a time to die;
a time to plant, and a time to uproot
the plant.
3 A time to kill, and a time to heal; a
time to tear down, and a time to build.
4 A time to weep, and a time to laugh; a
time to mourn, and a time to dance.
5 A time to scatter stones, and a time
to gather them; a time to embrace, and
a time to be far from embraces.
6 A time to seek, and a time to lose; a
time to keep, and a time to cast away.
7 A time to rend, and a time to sew; a
time to be silent, and a time to speak.
8 A time to love, and a time to hate; a
time of war, and a time of peace.

Eccles. 3, 1–8

47 I will show you what someone is like
who comes to me, listens to my words,
and acts on them.
48 That one is like a person building a
house, who dug deeply and laid the
foundation on rock; when the flood
came, the river burst against that house
but could not shake it because it had
been well built.
49 But the one who listens and does not
act is like a person who built a house on
the ground without a foundation. When
the river burst against it, it collapsed at
once and was completely destroyed.

Lk. 6, 47–49

Acknowledgements

Another slice of life has been spent. The last three years have been the most troublesome of my life; yet, the most fruitful ones by far. I have been rebuilding myself from the ground up, which has made me feel extremely lost and desperately needy of new landing places. Eventually, I have realised how precious this time could have become; I was not lingering on the sand, but digging deeper to lay foundations on rock. I am grateful for all the people around me: they have allowed me to face such a turbulent process without collapsing.

Firstly, I owe my deepest and most sincere gratitude to Prof. Giorgio Giacinto. He has been a solid pillar and, ultimately, the best supervisor I could ever wish for. Specifically, I admire his day-to-day work and effort to enable continuous new, fulfilling opportunities, along with careful, respectful, and wise support. I also acknowledge his excellent scientific profile, which greatly contributes to the success of PRA Lab; in this lab, I have found a wonderful mix of high competence and warmth.

In this sense, I am pleased to thank Dr. Davide Maiorca. He has been an inspiration since the days of my Master's thesis. I applaud his remarkable gift of bringing enthusiasm to each project, especially those oriented to students. For this reason, I wish him all the best for his career, as I am sure he will contribute to raising the level of the lab even further.

Moreover, I am grateful for spending a research period in another great lab at TU Braunschweig, Germany. I thank Prof. Konrad Rieck and all the group for hosting me. I found there excellent scientists that have allowed me to increase my skills, together with warm hospitality that has made my stay incredibly enjoyable.

Besides thanking Dr. Battista Biggio and all the other PRA Lab's senior components for their fruitful collaboration and suggestions, I thank all my Ph.D. colleagues. I will carry with pleasure all the days spent together exchanging opinions, collaborating, and having fun.

A significant part of my gratitude goes towards my friends and family. It is always fabulous and comforting to share openly with friends both our achievements and struggles. It has been crucial for me to talk and have fun with all of them, including the distant ones.

Finally, a special thanks to my family. I constantly feel amazed and blessed due to their endless support and trust towards me, so much so that I feel embarrassed as well: I cannot keep pace with the love I get from my parents, brothers, and sisters-in-law. Naturally, a super, sweet thank is devoted to my wonderful nieces Martina and Gioia, along with my *speedy* godson Nicola. I would never stop playing with them.

With renewed eyes and enriched by incredible experiences and the unique mindset a Ph.D. is able to offer, I hope to give back the love I receive.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The impressive growth and pervasiveness of *information and communication technology* in society is a known fact, and it will continue in the next few years. As an example, the European agency for cybersecurity (ENISA) prefigures a 2025 (European) scenario where [1]:

> *Devices are connected to the internet and have permeated everywhere. All the essential operators in all sectors (i.e. energy, transport, banks, digital infrastructure, hospitals), in all public administrations and across the industry are providing connected services. 80 billion devices (10 per person on the planet) are connected through the internet and the quantities of data produced have been doubling every 2 years.*

Such a connected society implies several impactful cybersecurity threats and the need for considerable efforts to address the emerging challenges. As confirmed by Europol [2], 'as time passes, the cyber-element of cybercrime infiltrates nearly every area of criminal activity'. As a consequence, recent decision ([3]) by the Members of the European Parliament (MEPs) is the adoption of the 'EU Cybersecurity Act', which establishes an EU-wide certification scheme to ensure that critical infrastructures (e.g. 5G), products, processes, and services meet cybersecurity standards. Moreover, ENISA has received a permanent mandate, together with more human and financial resources. Accordingly, increasing research efforts on cybersecurity will be spent. ENISA identifies the main topics where future research should be focused; they are the following ones [1]:

- **Awareness building:** it is making society aware of the impact and risks of technological change;
- **Capacity building:** it is refreshing education in order to fulfil the need of cybersecurity experts;
- **Existential threats:** these are trends and technologies that can be both an opportunity and a risk, such as artificial intelligence, quantum technologies, cybercrime, privacy.

Moreover, in a more recent report, ENISA highlighted the need to investigate further the usage of Machine Learning (ML) in Cyber Threat Intelligence [4]. In this sense, this thesis proposes to study the design of machine learning-based systems in the context of cybersecurity, with a particular focus on the detection of malware, as I discuss in the section that follows.

1.1 Malware and Ransomware

Among the several types of threats in the wild, malware attacks are steadily the most common ones. Specifically, if we look (Figure 1.1) at the trend of the past five years depicted by ENISA with its top-15 Threat Landscape [5–9], we can observe that malware always represents the top threat. Within this kind of threat, ransomware (which is considered as a different category by ENISA) has been representing a rising, dangerous threat in the past few years. Ransomware represents a serious threat that acts by locking the compromised system or encrypting its data, then forcing the owners to pay a ransom in order to restore the system functionality. Despite the increasing diffusion of cloud-based technologies, users still store the majority of their data directly on local systems. For this reason, such attacks are particularly devastating, as they could destroy sensitive data of private users and companies (which often neglect to make backups of sensitive data). In this sense, Figure 1.1 shows its concerning increase until 2017, which has motivated the special attention of this thesis to this kind of threat. Although 2018 and 2019 saw a decrease in the ENISA ranking (the 2020 report points out a new increasing trend though [9]), the strategy for such attacks is precisely becoming more and more targeted. For example, 2019 saw a 365% increase of attacks against businesses with respect to 2018, which may explain the 48.5% increase in the amount of paid ransoms (a total of 10.1 billion USD) [10].

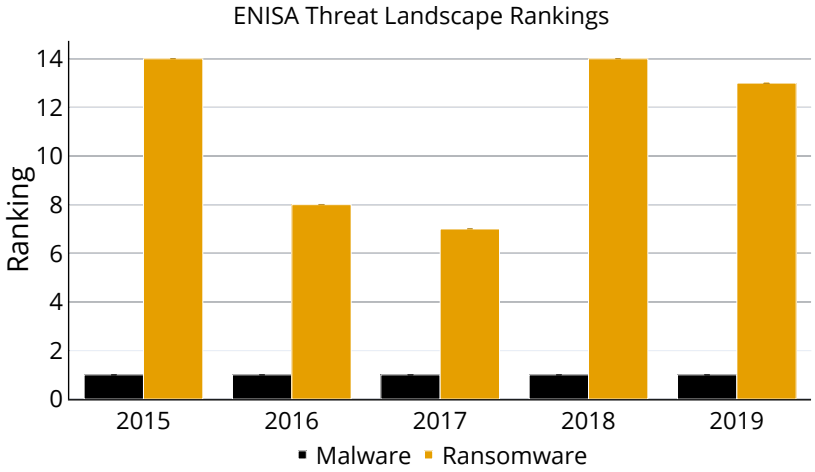


FIGURE 1.1: ENISA top threats ranking (out of 15 threats) of the last five years for the malware and ransomware categories. The lower the ranking value, the more relevant the threat. Data elaborated from [5–9].

To combat such kind of threats, which also affect the Android domain (as I discuss in Section 3.2), the security community strives to develop effective defence tools, such as malware detectors. In this sense, malware detection is one of the areas where machine learning is successfully employed due to its high discriminating power and the capability of identifying novel variants of malware samples. Typically, the problem formulation is strictly correlated to the use of a wide variety of features covering several characteristics of the entities to classify. Apparently, this practice provides considerable detection performance. However, *it hardly permits us to gain insights into the knowledge extracted by the learning algorithm*. This issue is crucial since it negatively influences two main aspects of detectors:

- **Effectiveness:** detectors might learn spurious patterns, i.e. they make decisions possibly on the basis of artefacts or non-descriptive elements of the input samples;
- **Security:** the learning algorithms might become particularly vulnerable to *adversarial* attacks, i.e. carefully-perturbed inputs that overturn their outputs.

Both of these concerns undermine the actual use of such detectors in real working environments. Consequently, this issue sounds an alarm and highlights the necessity to *(i)* improve the design process of detection systems and *(ii)* make them tailored to the specific peculiarities of the attacks to detect.
Within malware detection, *Android ransomware* represents a challenging yet illustrative domain for assessing the relevance of the considerations above. As a matter of fact, the attackers typically develop such dangerous apps so that normally-legitimate components and functionalities perform malicious behaviour; thus, making them harder to be distinguished from genuine applications. In this sense, adopting a well-defined variety of features and relying on some kind of explanations about the logic behind such detectors could improve their design process since it could reveal truly characterising features; hence, guiding the human expert towards the understanding of the most relevant attack patterns. Moreover, new legislation may enforce ML-based systems to be interpretable enough to provide an explanation of their decisions, as they can have significant consequences depending on the domain, such as the safety-critical cases of medical diagnoses or autonomous driving [11]. For instance, the Recital 71 of the EU GDPR[1] states:

> *The data subject should have the right not to be subject to a decision, which may include a measure, evaluating personal aspects relating to him or her which is based solely on automated processing and which produces legal effects concerning him or her or similarly significantly affects him or her, such as automatic refusal of an online credit application or e-recruiting practices without any human intervention. [...]*
> *In any case, such processing should be subject to suitable safeguards, which should include specific information to the data subject and the* ***right*** *to obtain human intervention, to express his or her point of view,* ***to obtain an explanation of the decision*** *reached after such assessment and to challenge the decision.*

For all these reasons, research on the so-called *explainable machine learning* is currently rising, and it represents one of the tools used throughout the work of this thesis to counteract the above-discussed issues.

1.2 Contribution

Given the above-described context, the ultimate goal of the thesis is to explore strategies that may improve the design process of malware detectors. In particular, the main contribution of this work is aimed at addressing concerns about the detectors' *effectiveness*, and can be summarised as follows:

[1] `https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32016R0679&from=EN`

1. Identification of a set of effective features for Android ransomware detection, along with their evaluation through different security-specific metrics. In this sense, while most of the systems in the literature rely on different types of information extracted from multiple parts of the apps, *this work starts by considering a limited number of features of the same type*;

2. Proposal of a method to integrate explainability techniques into the identification and assessment of features characterising malware behaviour. This way, it could be possible to *broaden the variety in the feature set with an attentive approach.*

Moreover, I co-authored work that intends to address the *security* of detectors and provides the following contribution:

1. Proposal of a proxy metric based on explanations, which reveals models' robustness to adversarial attacks;

2. Evaluation through explanations of the practical feasibility for attackers to alter Android apps in order to evade classifiers.

Ultimately, this work highlights the necessity to adopt a design process that is aware of the weaknesses and attacks against ML-based detectors, and proposes explainability techniques as one of the tools to counteract them.

1.3 Organisation

In this thesis, I first frame more extensively the challenges resulting from the usage of machine learning for cybersecurity problems (Chapter 2). Thereafter, I focus on the target domain of the thesis, i.e. Android malware detection. In particular, after providing background notions on Android security (Chapter 3), I present the work through two main pathways. The first one (Chapter 4) focuses on identifying the main traits that result to be characterising and effective for Android ransomware detection. Then, explainability techniques are used to propose methods to assess the validity of the considered features.
The second pathway (Chapter 5) broadens the view by exploring the relationship between explainable machine learning and adversarial attacks. I first illustrate work aiming to define metrics extracted from explainability techniques that can reveal models' robustness to adversarial attacks. I then move to an assessment of the practical feasibility for attackers to alter the features that affect models' output the most.
Finally, I discuss the findings and limitations of the thesis, along with potential future work (Chapter 6).

Chapter 2

Challenges in Cybersecurity: Adversarial and Explainable Machine Learning

This chapter aims to introduce two of the main topics that characterise research on machine learning, namely adversarial learning (Section 2.1) and explainable machine learning (Section 2.2). Before jumping into them, it is worth discussing briefly the general background of machine learning for cybersecurity tasks.
As hinted in Chapter 1, machine learning systems are nowadays being extensively adopted in computer security applications, such as network intrusion and malware detection, as they have obtained remarkable performance even against the increasing complexity of modern attacks [12–14]. More recently, learning-based techniques based on static analysis have proven to be especially effective at detecting Android malware, which constitutes one of the major threats in mobile security. In particular, these approaches have shown great accuracy even when traditional code concealing techniques (such as static obfuscation) are employed (see Chapter 3) [15–18].

Despite the successful results reported by such approaches, the problem of designing effective malware detectors — or solving different cybersecurity tasks — through machine learning is still far from being solved, and it does not pertain only to adversarial attacks and the lack of interpretability. As a matter of fact, several other pitfalls may undermine the validity of such systems. In the process of working on ML-based malware detection, I have been recognising more and more challenges and caveats to pay attention to. Therefore, with no claim for this thesis to be immune from those issues, it is worth encouraging awareness in this direction as it may also help to catch the potential role that explainable machine learning may have to address some of them.

Pitfalls in ML for cybersecurity To illustrate some of the risks designers and researchers may fall into when using ML for cybersecurity, it is possible to follow the recent work by Arp et al. [19], who have identified and systematised ten weaknesses, spanning from design choices to performance evaluation and practical deployment. Each paper out of the 30 they have analysed from top-tier security conferences within the past ten years exhibits at least three of them.
Among them, a first pitfall is represented by *false causality*, i.e. the situation where the learning model adapts to artefacts unrelated to the security problem under

inspection; thus, not solving it in practice. This issue is part of the motivation for the work of this thesis, and explainable machine learning is definitely one of the tools to limit it. Another pitfall consists of evaluating models with an *inappropriate baseline*, i.e. without performing a proper comparison with other state-of-the-art approaches. Notably, this also includes avoiding a comparison with non-ML-based systems. Similarly, typical proposals for novel models are assessed *lab-only*, i.e. the test of the system is only performed *in vitro*, with no discussion of the practical limitations. As the last point — I refer the reader to [19] for a complete disquisition of the pitfalls — the usage of an *inappropriate threat model* that does not consider the security of the algorithms is another weakness. Notably, a proper threat model should also examine the practical capabilities of attackers, as discussed in Chapter 5.

2.1 Adversarial Machine Learning

According to a recent survey by Biggio and Roli [20], several works questioned the security of machine learning since 2004. Two pioneering works were proposed by Dalvi et al. [21] in 2004 and by Lowd and Meek [22] in 2005. Those works, considering linear classifiers employed to perform spam filtering, have demonstrated that an attacker could easily deceive the classifier at test time by performing a limited amount of carefully-crafted changes to an email. Subsequent works have proposed attacker models and frameworks that are still used to study the security of learning-based systems also against training-time attacks [23–25]. The first gradient-based poisoning [26] and evasion [27] attacks have been proposed by Biggio et al., respectively, in 2012 and 2013. Notably, in [27] the authors have also introduced two important concepts that are still heavily used in the adversarial field: *high-confidence* adversarial examples and a *surrogate* model. This work anticipated the discovery of the so-called *adversarial examples* against deep neural networks [28, 29].

The vulnerability to evasion attacks has also been studied on learning systems designed to detect malware samples (for example, on PDF files [30, 31]); hence, raising serious concerns about their usability under adversarial environments. In particular, for Android malware detectors, Demontis et al. [17] have demonstrated that linear models trained on the (static) features extracted by Drebin (see Section 5.1.1) can be easily evaded by performing a fine-grained injection of information by employing gradient descent-based approaches. Grosse et al. [32] have also attained a significant evasion rate on a neural network trained with the Drebin feature set. Although the adversarial robustness of other Android detectors aside from [15] has not been fully explored, it is evident that employing information that can be easily injected or modified may increase the probability of the attacker to attain successful evasion. In a sense, this aspect also motivates the work described in Section 5.2.
To better understand the main characteristics that adversarial attacks may assume and the settings of the works in Chapter 5, in the following, I briefly illustrate the typical threat model schema.

2.1.1 Modelling Threats

The main aspects to consider when modelling the adversarial threats are the following: the *attacker's goal*, *knowledge*, and *capability* [20, 27].

Attacker's goal The goal of an attack may be dissected according to different aspects. One consists of pursuing a *generic* or *specific* attack. In the first case, the attacker is generically interested in having the samples misclassified. In the second case, its goal is to have specific samples classified as a target class. For example, in a malware detector, the attackers aim to make the system classify malware apps as trusted ones. Consequently, the attack could involve any kind of sample (*indiscriminate*) or a specific subset (*targeted*). Ultimately, the attackers may want to compromise the system's *availability*, *privacy*, or *integrity*.

Attacker's knowledge Given a knowledge parameter $\boldsymbol{\theta}$ that indicates the amount of information about the target detection system available to attackers, this can be related to *(i)* the dataset D, *(ii)* the feature space X, *(iii)* the classification function f, and *(iv)* the classifier's hyperparameters $\boldsymbol{w}$. Accordingly, $\boldsymbol{\theta} = (D, X, f, \boldsymbol{w})$ corresponds to the scenario of *perfect knowledge* about the system and represents the worst case for a defender. It is unlikely to see such a scenario in real cases, as attackers often have incomplete information (or no information at all) about the target system. However, it can serve as a worst-case scenario to assess the maximum vulnerability of a system. More realistic settings (*limited-knowledge*) do not have complete information about the target system, as in the case of the scenario considered in Section 5.2.

Attacker's capability This aspect first refers to the possibility for the attackers to manipulate training and test data. When they are able to act on both of them, we talk about *poisoning* attacks, while altering samples at test time corresponds to *evasion attacks*. In this thesis, I only consider evasion attacks.
Moreover, attackers are often constrained in the kind of modifications that they are able to perform on a sample, e.g. a malicious Android application. For example, the Android detection domain mostly allows the attacker to only add new elements to the app (*feature addition*), but does not permit removing them (*feature removal*).

2.2 Explainable Machine Learning

Chapter 1 has provided a set of motivations and goals that had brought to identifying 'explainable machine learning' as a research field worth exploring. However, such a topic appears to be extremely vast, so that even defining it turns out to be not as trivial. Consequently, this section proposes to *(i)* clarify the terms used in the scope of this thesis (e.g. 'explainable machine learning'), *(ii)* provide the essential elements that illustrate the potential directions of this field, and *(iii)* list — with no claim of completeness — popular explanation techniques and present the ones employed in this thesis (Section 2.2.1).

As for the first point, the field as a whole is usually indicated as 'Interpretable Artificial Intelligence', 'Explainable Artificial Intelligence', or 'XAI'. To the best of my knowledge, there is no agreement on which of them is the most used within the research community, as surveys come up with contrasting conclusions [33, 34]. Despite being less frequent, in this thesis, the term 'explainable machine learning' is used, as the focus is specifically on machine learning. Concerning more specialised definitions, the literature embodies terms such as 'explainability', 'interpretability',

'understandability', 'comprehensibility', 'transparency' [33, 34]. Each one, assuming the capability of establishing a common ground, reflects certain peculiarities in terms of several aspects, such as motivations, goals, procedures. With this respect, the approach of this thesis is not to provide formal definitions, but to detail motivations, context, and goals for each work. However, in order to have a basic reference, I follow this broad distinction by Barredo Arrieta et al. [34]:

> *'Interpretability' refers to a passive characteristic of a model referring to the level at which a given model makes sense for a human observer. This feature is also expressed as 'transparency'. By contrast, 'explainability' can be viewed as an active characteristic of a model, denoting any action or procedure taken by a model with the intent of clarifying or detailing its internal functions.*

The focus of the thesis is then on 'explainability', rather than 'interpretability'.

Moving to a simple taxonomy that can summarise explainable machine learning, the criteria to consider could be: *why* explanations are produced, *whom* they are addressed to, *what* they consist of. In the following, I sum up such basic aspects to the extent that is needed for this dissertation:

Goals As Doshi-Velez and Kim [35] have pointed out, not all ML systems require interpretability. Rather, it is needed when the formalisation of a problem is *incomplete*. For example, we may produce explanations for *knowledge extraction* since we do not have a full understanding of the mechanisms of the problem under analysis. If we want to validate models, explanations can be used to perform *debugging*; this way, it is possible to find unexpected behaviour and the conditions under which the model fails. Moreover, *legal or ethical requirements* are becoming more and more pressing. For example, certain domains are particularly sensitive in ensuring no biases are present in the model, e.g. racial ones in the context of *fairness*.

Stakeholders Deriving from the above goals, explanations can be addressed to different stakeholders, typically identified as [36]:

- **Designers and theorists:** they may want to improve the model design or the algorithms, in order to fix their weaknesses or extract knowledge from them;
- **Ethicists:** they need explanations to ensure models are *fair*;
- **End users:** explanations are welcome to gain *trust* about the decisions made by the models.

To better clarify the concrete significance of goals and stakeholders of explanations, we can consider the case of smart vehicles, of which I have been analysing their security and the potential impact of explainable machine learning to them [11, 37]. Smart vehicles can be seen both as *human-agent systems* — highlighting the interaction between digital systems and human users — and as *cyber-physical systems* — highlighting the interaction between cyberspace and the physical environment.
From the first point of view, the interaction of the system is directed to the end-user. Consequently, the reason to produce and present explanations is gaining trust in the decisions adopted by the system. Since a modern vehicle ecosystem includes both safety- and non-safety-critical ML-based algorithms, for the former ones, it is crucial

to *(i)* provide explanations, *(ii)* produce them on top of *transparent* (see next paragraph) models, as they are able to provide direct and faithful explanations [38]. In the second case, having explanations could be beneficial but not strictly necessary; hence, making the deployment of explainable algorithms optional. Recent work started exploring more precisely both the context and the necessity for explainable algorithms in human-agent systems. For example, Glomsrud et al. [39] have studied the context of autonomous vessels. Besides developers and assurance figures, the main human interactions, in this case, regard passengers and external agents. The former ones may want to have feedback from the vessel (e.g. a ferry) during boarding, docking, or abnormal situations, in order to be warned about the travel status or potential danger and prevent panic. External actors could be swimmers, kayakers or boats that are close to the vessel; hence, they need to understand its intentions as early as possible.

Explanation generation In this respect, different criteria can categorise explanation techniques:

- ***Post-hoc* and *interpretable-by-design* cases:** the first one refers to the act of explaining a model with external, specific techniques, while the second one extract explanations inherently from the model, which is then considered as *transparent* by design. This distinction recalls and is related to the definitions of 'explainability' and 'interpretability' mentioned above;
- ***Black-box* and *white-box* techniques:** since models could be not fully accessible (e.g. for preserving intellectual property), black-box techniques propose to infer explanations only from models' outputs, as opposed to *white-box* techniques that can leverage the inner working of the algorithms;
- ***Local* and *global* techniques:** it is possible to interpret a single decision of the model or its whole logic;
- ***Feature attribution* and *high-level concepts*:** the typical output of an explanation consists of a numerical value for each feature, which represents its relevance to the scope of the single classification or model logic. However, recent work has started exploring the capability of identifying higher-level concepts [40, 41].

2.2.1 Explanation Techniques

Several approaches for interpretability have been proposed, with particular attention to *post-hoc* explanations for black-box models. In the following, I briefly describe the prominent explainability methodologies proposed in this sense. In 2016, Ribeiro et al. [42] proposed LIME, a model-agnostic technique that provides local explanations by generating small perturbations of the input sample; thus, obtaining the explanations from a linear model fitted on the perturbed space. Lundberg et al. [43] have unified different techniques, including LIME, under the name of SHAP; by leveraging cooperative game theory results, they identify theoretically-sound explanation methods and provide feature importance for each prediction.
Koh and Liang [44] have shown that using a gradient-based technique called *influence functions*, which is well known in the field of robust statistics, it is possible to associate each input sample to the training samples (*prototypes*) that are most responsible for its prediction. The theory behind the techniques proposed by the

authors holds only for classifiers with differentiable loss functions. However, the authors have empirically shown that their technique provides sensible prototypes also for classifiers with non-differentiable losses if computed on a smoothed counterpart. Finally, Guo et al. [45] have proposed LEMNA, a method specifically designed for security tasks, i.e. that is optimised for RNN and MLP networks, and that highlights the feature dependence (e.g. for binary code analysis).

2.2.1.1 Gradient-based Explanation Methods

One of the characteristics common to several explanation techniques is deriving them from the calculation of the gradient of the decision function with respect to the input. In particular, the ones considered in this thesis are usually referred to as 'gradient-based attribution methods', where 'attribution' means the contribution of each input feature to the prediction of a specific sample. In a two-class setting, the positive (negative) value of an attribution indicates that the classifier considers the corresponding feature as peculiar of the positive (negative) class. In the following, I briefly describe three gradient-based techniques.

Gradient The simplest method to obtain attributions is to compute the gradient of the discriminant function f with respect to the input sample $\boldsymbol{x}$. For image recognition models, it corresponds to the saliency map of the image [46]. The attribution of the i^{th} feature is computed as:

$$\text{Gradient}_i(\boldsymbol{x}) := \frac{\partial f(\boldsymbol{x})}{\partial x_i} \tag{2.1}$$

Gradient*Input This technique has been proposed by Shrikumar et al. [47]. This approach is more suitable than the previously-proposed ones when the feature vectors are sparse. For example, the previously proposed approaches [42, 46] tended to assign relevance to features whose corresponding components are *not* present in the considered application; thus, making the corresponding predictions challenging to interpret. To overcome this issue, Gradient*Input leverages the notion of *directional derivative*. Given the input point $\boldsymbol{x}$, it projects the gradient $\nabla f(\boldsymbol{x})$ onto $\boldsymbol{x}$, to ensure that only the non-null features are considered as relevant for the decision. More formally, the i^{th} attribution is computed as:

$$\text{Gradient*Input}_i(\boldsymbol{x}) := \frac{\partial f(\boldsymbol{x})}{\partial x_i} * x_i \tag{2.2}$$

Integrated Gradients Sundararajan et al. [48] have identified two axioms that attribution methods should satisfy: *implementation invariance* and *sensitivity*. Accordingly to the first one, the attributions should always be identical for two functionally equivalent networks, e.g. they should be invariant to the differences in the training hyperparameters, which lead the network to learn the same function. The second axiom is satisfied if, for every input predicted differently from a baseline (a reference vector that models the neutral input, e.g. a black image) and that differs from the baseline in only one feature, has, for that feature, a non-zero attribution. In the same paper, they have proposed a gradient-based explanation called Integrated

Gradients that satisfies the axioms explained above. This method, firstly, considers the straight-line path from the baseline to the input sample and computes the gradients at all points along the path. Then, it obtains the attribution by accumulating those gradients. The attribution along the i^{th} dimension for an input $\boldsymbol{x}$ and baseline $\boldsymbol{x}'$ is defined as:

$$\text{IntegratedGrads}_i(\boldsymbol{x}) := (x_i - x_i') \cdot \int_{\alpha=0}^{1} \frac{\partial f\left(\boldsymbol{x}' + \alpha \cdot (\boldsymbol{x} - \boldsymbol{x}')\right)}{\partial x_i} d\alpha \tag{2.3}$$

To efficiently approximate the previous integral, one can sum the gradients computed at p fixed intervals along the joining path from $\boldsymbol{x}'$ to the input $\boldsymbol{x}$:

$$\text{IntegratedGrads}_i^{\text{approx}}(\boldsymbol{x}) := (x_i - x_i') \cdot \sum_{k=1}^{p} \frac{\partial f\left(\boldsymbol{x}' + \frac{k}{p} \cdot (\boldsymbol{x} - \boldsymbol{x}')\right)}{\partial x_i} \cdot \frac{1}{p} \tag{2.4}$$

For linear classifiers, where $\partial f / \partial x_i = w_i$, this method is equivalent to Gradient*Input if $\boldsymbol{x}' = \mathbf{0}$ is used as a baseline, which is a well-suited choice in many applications [48]. Therefore, in this particular case, also the Gradient*Input method satisfies the above-mentioned axioms.

Chapter 3

Android Security

This chapter provides essential background about Android and its security. I first summarise the main components of the operating system (Section 3.1). I then depict the scenario of Android malware attacks, especially ransomware ones (Section 3.2), and conclude with an overview of methods to analyse and detect them (Section 3.3).

3.1 Background on Android

Android applications are zipped `.apk` — i.e. `APK`, Android Application Package — archives that contain the following elements: *(i)* The `AndroidManifest.xml` file, which provides the application package name, and lists its basic components, along with the permissions that are required for specific operations; *(ii)* One or more `classes.dex` files, which are the true executable of the application, and which contain all the implemented classes and methods (in the form of `Dalvik` bytecode) that are executed by the app. This file can be disassembled to a simplified format called `smali`; *(iii)* Various `.xml` files that characterise the application layout; *(iv)* External resources that include, among others, images and native libraries.

Android applications are typically written in `Java` or `Kotlin`, and they are compiled to an intermediate bytecode format called `Dalvik` — which will also be referred to as `Dex` bytecode or `dexcode` — whose instructions are contained in the `classes.dex` file. This file is then further parsed at install time and converted to native ARM code that is executed by the Android RunTime (`ART`). This technique allows to greatly speed up execution in comparison to the previous runtime (`dalvikvm`, available till Android 4.4), in which applications were executed with a just-in-time approach (during installation, the `classes.dex` file was only slightly optimised, but not converted to native code).
As the elements inspected in the scope of this thesis are the `Dex` bytecode and the `manifest`, they are described with more details below.

Android Manifest This file contains core information about the Android application, such as its package name or the supported API levels. It lists the app's *components*, i.e. the elements that define its structure and functionalities. For example, the screens visualised by the user are built upon an `activity`; a background task is executed through a `service`. App components can also listen to specific events and be executed in response to them (`receiver`). This applies to developer-defined events or system ones, such as a change in the device's connectivity status

(`CONNECTIVITY_CHANGE`) or the opening of an application (`LAUNCHER`). Special types of components are *entry points*, i.e. activities, services, and receivers that are loaded when requested by a specific filtered intent (e.g. an activity is loaded when an application is launched, and a service is activated when the device is turned on). One of the most important sections of the `manifest` comes from the listing of the *permissions* used by the application (`uses-permission` tag). As a matter of fact, the app developer has to ask the user the right to use certain functionalities. These can be related to the device (e.g. `ACCESS_NETWORK_STATE`) or functionalities (e.g. `SEND_SMS`). In this thesis, permissions are always indicated in capital letters.

Dex bytecode The `classes.dex` file embeds the compiled source code of the applications, including all the user-implemented methods and classes. For example, it may contain specific API calls that can access sensitive resources such as personal contacts (*suspicious calls*). Additionally, it contains all system-related, *restricted API calls* that require specific permissions (e.g. writing to the device's storage). The main components of a `.dex` file are:

- **Header:** this contains information about the file composition, such as the offsets and the size of other parts of the file (e.g. constants and data structures). This data collection is crucial to reconstruct the bytecode in the correct way when the code is compiled to ARM;
- **Constants:** they represent the addresses of the strings, flags, variables, classes, and method names of the application;
- **Classes:** this is the definition of all the class parameters, like the super class, the access type, and the list of methods with all the references to the data contained in the data structure;
- **Data structure:** this is the container for the actual data of the application, such as the method code or the content of static variables.

3.2 Android Malware

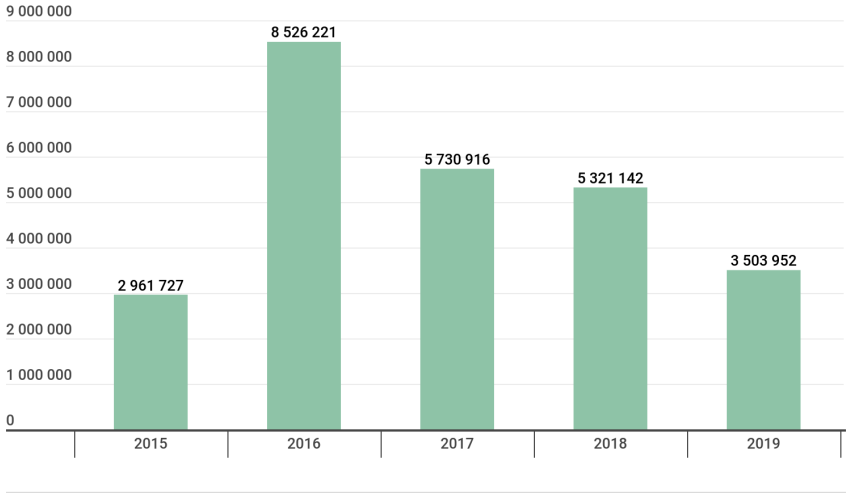
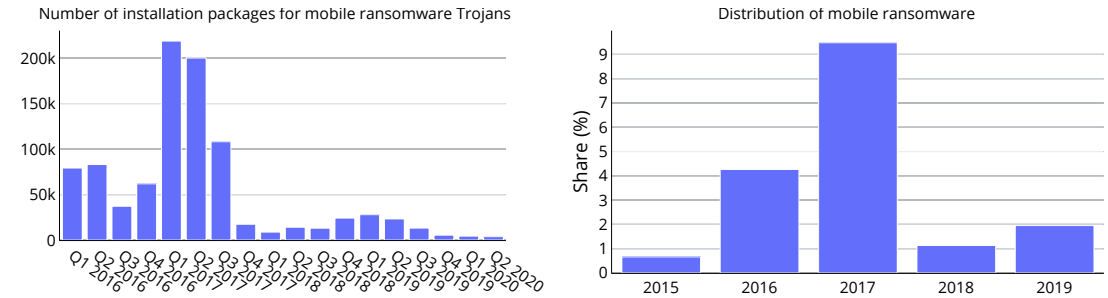


FIGURE 3.1: Number of mobile malicious installation packages on Android [49].

Android has always been the top target of mobile malware attacks. To counteract such offensives, several security improvements have been made to Android over the past few years. Both Kaspersky's (Figure 3.1) and Trend Micro's data show a declining trend of malicious installation packages since 2016, which seems to indicate defence measures result to be effective (this will be discussed further in the next section) [49, 50]. However, the context of mobile threats is highly dynamic. For example, Kaspersky [49] points out the appearance of several highly sophisticated mobile banking threats in 2019, and Trend Micro [50] underlines the increasing sophistication in targeted campaigns. To better understand such a trend, the section that follows depicts the scenario for ransomware attacks, along with their characteristics.

3.2.1 Ransomware

Similar to the trend illustrated in Section 1.1, mobile ransomware has apparently reached its peak. More specifically, Figure 3.2 shows the steep rise of ransomware from 2015 to 2017, in terms of the both number of installation packages and distribution share over all types of mobile threats. This fact had motivated the focus of most of the thesis' work on ransomware. After that, we can observe a significant decrease in infections. As of September 2020 and according to Kaspersky [51], such a trend is motivated by two reasons: *(i)* it is much harder to extort cash from users than to steal the bank account data; thus, making the device infection more costly, and *(ii)* ransomware is a threat the user likely wants to fight in order to restore his device, even by factory-resetting the device — except in the case of encrypted files. Another reason I point out is the progress of security measures against such attacks. In this sense, Kaspersky [51] highlights that most of the current spread families date back from 2017. This fact could indicate attacks target older versions of Android since — as of January 2021 — the majority of devices run Android Oreo (API level 27, released in August 2017) or lower. [1]



(a) (b)

FIGURE 3.2: Number of installation packages (a) and share of distribution over all malware types (b) for mobile ransomware. Data elaborated from [49, 51–54].

To better understand the security improvements of Android, it is worth illustrating how typical ransomware attacks work. The majority of ransomware attacks for Android are based on the goal of *locking* the device screen. In this case, attackers typically take the following strategy: they create an `activity` upon which

[1] Data gathered from Android Studio's Android Platform Version Distribution.

a non-dismissable window is shown. This `activity` is forced to stay always in the foreground, and it can be restarted when killed or the device gets rebooted. Moreover, they disable the navigation buttons (e.g. the *back* functionality). Newer versions of Android, primarily since API level 28 (Android Pie), have implemented countermeasures in response to this strategy. For example, *(i)* the device can be booted in *Safe Mode*, where the system blocks third-party apps from running, *(ii)* starting an activity from the background is not possible without leveraging notifications, and *(iii)* the status bar takes priority and shows a specific notification that allows disabling overlay screens from the running app. However, as such recent Android versions do not reach the totality of the devices, locking behaviour remains a relevant threat.
Locking is generally preferred to the data encryption strategy because it does not require operating on high-privileged data. There are, however, *crypto*-ransomware apps that perform data encryption. Examples of crypto families are: `Simplocker`, `Koler`, `Cokri`, and `Fobus`. In this case, the attacker shows a window that could not necessarily be constantly displayed, because his main focus is to perform encryption of the user data (e.g. photos, videos, documents).
In both cases, the created window includes a threatening message that instructs the user to pay the ransom, which will theoretically permit to *(i)* suppress the permanent screen (locking case) or *(ii)* decipher the data (crypto case).

As locking and encryption actions require the use of multiple functions that involve core functionalities of the system (e.g. managing entire arrays of bytes, displaying activities, manipulating buttons and so on), *attackers tend to use functions that directly belong to the Android System API*. It would be extremely time consuming and inefficient to build new APIs that perform the same actions as the original ones. This motivates the design choices of the work in Section 4.2.
As an example of ransomware behaviours, consider the `dexcode` snippet provided by Listing 3.1, belonging to *locker-type* ransomware[2]. In this example, it is possible to observe that the two function calls (expressed by `invoke-virtual` instructions) that are actually used to lock the screen (`lockNow`) and reset the password (`resetPassword`) are System API calls, belonging to the class `DevicePolicyManager` and to the package `android.app.admin`.

```
1
2 invoke-virtual {v9}, Landroid/app/admin/DevicePolicyManager;->lockNow()V
3 move-object v9, v0
4 move-object v10, v1
5
6 ...
7
8 move-result-object v9
9 move-object v10, v7
10 const/4 v11, 0x0
11 invoke-virtual {v9, v10, v11}, Landroid/app/admin/DevicePolicyManager;->
    resetPassword(Ljava/lang/String;I)Z
```

Listing 3.1: Part of the onPasswordChanged() method belonging to a locker-type ransomware sample.

The same behaviour is provided by Listing 3.2, which shows the encryption

[2]MD5: `0cdb7171bcd94ab5ef8b4d461afc446c`

function employed by a *crypto-type* ransomware sample[3]. Again, the functions to manipulate the bytes to be encrypted belong to the System API (`read` and `close`, belonging to the `FileInputStream` class of the `java.io` package; `flush` and `close`, belonging to the `CipherOutputStream` class of the `javax.crypto` package).

```
1
2     Ljava/io/FileInputStream;->read([B)I
3     move-result v0
4     const/4 v5, -0x1
5     if-ne v0, v5, :cond_0
6     invoke-virtual {v1}, Ljavax/crypto/CipherOutputStream;->flush()V
7     invoke-virtual {v1}, Ljavax/crypto/CipherOutputStream;->close()V
8     invoke-virtual {v3}, Ljava/io/FileInputStream;->close()V
```

Listing 3.2: Parts of the encrypt() method belonging to an encryption-type ransomware sample.

3.3 Malware Analysis and Detection

This section makes a brief, non-exhaustive overview of the methods developed to perform Android malware detection. In particular, I illustrate their high-level approach (Section 3.3.1), the typical employed features (Section 3.3.2), and the strategies attackers devise to bypass detection (Section 3.3.3) [55, 56].

3.3.1 Approaches

A first relevant aspect for the different approaches to design detectors is the distinction between systems that perform *static*, *dynamic*, or *hybrid* analysis.

Static analysis is based on disassembling an Android application and scanning its components to find malicious traces without executing the application. Consequently, this approach has the advantage of being typically fast and resource-efficient. Different research works have been published about this type of analysis. Arzt et al. [57] have proposed FlowDroid, a security tool that performs static taint analysis within the single components of Android applications. Feng et al. [58] have proposed Apposcopy, another tool that combines static taint analysis and intent flow monitoring to produce a signature for applications. Garcia et al. [59] have proposed RevealDroid, a static system for detecting Android malware samples and classifying them in families. The system employs features extracted from reflective calls, native APIs, permissions, and many other characteristics of the file. The attained results showed that RevealDroid was able to attain very high accuracy, resilience to obfuscation. However, the number of extracted features can be extremely high and depends on the training data.

Concerning dynamic analysis, it is based on executing and monitoring an application in a controlled environment (i.e. sandbox or virtual machine). The goal is to inspect the interactions between the application and the system to reveal all the suspicious behaviours. Zhang et al. [60] have proposed VetDroid, a dynamic analysis platform to detect interactions between the application and the system through the

[3]MD5: `59909615d2977e0be29b3ab8707c903a`

monitoring of permission use behaviours. Tam et al. [61] have proposed CopperDroid, a dynamic analyser that aims to identify suspicious high-level behaviours of malicious Android applications. Chen et al. [62] have proposed RansomProber, a dynamic ransomware detection system that uses a collection of rules to examine several execution aspects, such as the presence of encryption or unusual layout structures. Dynamic analysis is more challenging to implement. It requires more computational resources and time to be executed. This is why this type of analysis cannot be implemented on a mobile device. However, it has better performances in detecting well-known and never seen malware families.

Finally, the straightforward third approach combines static and dynamic analyses to take advantage of their strengths and compensate for their weaknesses. An example in this sense is the work from Chen et al. [16], who have proposed StormDroid, a static and dynamic machine learning-based system that extracts information from API-calls, permissions, and behavioural features.

3.3.2 Features

As hinted with the above-mentioned approaches, several characteristics (that translate to features for ML-based systems) can be extracted to discriminate malware from legitimate apps. The main ones are the following:

- **Permissions:** Android requires the app developer to request permission to use a particular functionality expressly. Depending on its dangerousness, that permission can be granted automatically or after explicit user agreement. As an example, Arp et al. [15] have proposed Drebin, a machine learning system that uses static analysis to discriminate between generic malware and trusted files (also described in Section 5.1.1). They have extracted permissions, along with IP addresses, suspicious API calls, intents, and so forth;
- **Intents and events:** these are objects that *call* components of the same app, other apps, or system in order to perform specific actions (e.g. send emails, add a contact). Yang et al. [63] analysed malicious apps by defining and extracting the context related to security-sensitive events. In particular, the authors defined a model of context based on two elements: activation conditions (i.e. what makes specific events occur) and guarding conditions (i.e. the environmental attributes of a specific event);
- **Hardware components:** this refers to the apps' requests (in the `manifest`) to use specific hardware (e.g. GPS);
- **Function calls:** this may refer to generic function calls, or specific ones such as System API calls (discussed in Chapter 4);
- **Information flow:** this consists of monitoring the flow of information, which may be function calls, data, communication between processes. For example, Avdiienko et al. [64] have used taint analysis to detect anomalous flows of sensitive data, a technique that has allowed to detect novel malware samples without previous knowledge;
- **Dependency graphs:** these refer to providing a representation of the dependences between statements. For example, Zhang et al. [65, 66] have employed dependency graphs extracted by observing *network traffic*. In particular, in [65], they have proposed a traffic analysis method that employs scalable algorithms for the detection of malware activities on a host. Such

a detection was performed by exploring request-level traffic and the semantic relationships among network events. In [66], they have analysed information extracted from the dynamic profiling of the traffic generated by benign applications and modelled (through graphs called *triggering relation graphs*) the triggering relationship of the generated network events (i.e. how such events are related to each other) to identify anomalous, malicious ones.

3.3.3 Evasion Strategies

As the last point of view, it is worth highlighting that the above-mentioned detection strategies have to deal with different elusion techniques elaborated by attackers to avoid detection. Such stratagems undermine the effectiveness of both static and dynamic approaches. Firstly, a trivial elusion strategy is represented by *repackaging*, i.e. performing slight alterations to the `.apk` files, with the simple goal of evading signature-based detectors. A more relevant technique is *obfuscation*, which includes trivial as well as advanced strategies. In particular, it may consist of (as discussed in Section 4.2) *(i)* renaming methods, classes, and identifiers, *(ii)* employ *reflection* to load code dynamically, and *(iii)* encrypting code, such as classes. Moreover, it can also include control flow obfuscation, i.e. transforming method calls (typically by changing their order) without changing the app's semantics. Accordingly, all those types of elusion techniques are often able to overcome static analysis.
Additionally, dynamic analysis can also be complicated or prevented. In these cases, the analysed is typically performed in virtual environments. Consequently, malware may detect whether it is being executed within them and stop itself. Other stratagems include pausing the execution of malicious activities on specific time frames, events, or even depending on the motion sensors [67].

Chapter 4

Design of Android Ransomware Detectors

In this chapter, the focus is on the work done to design effective Android ransomware detectors. In particular, I first examine the choice of System API calls as features, in order to determine if they reveal to be both effective and resilient to attackers' anti-detection strategies (Section 4.2, based on the work by Scalas et al. [18]). Then, I focus on proposing a method to leverage gradient-based explanation techniques in order to understand if features (System API calls and permissions) really characterise ransomware behaviour (Section 4.3, based on the work by Scalas et al. [68]). Before diving into these works, a brief overview of the literature on Android ransomware detectors is provided (Section 4.1).

4.1 Related Work

Most Android malware detectors typically discriminate between malicious and benign apps, and are referred to as *generic malware-oriented* detectors (see Section 3.3). However, as the scope of the works in this chapter is oriented to ransomware detection, this section will be focused on describing systems that aim to detect such attacks (*ransomware-oriented* detectors) specifically.

The most popular *ransomware-oriented* detector is `HelDroid`, proposed by Andronio et al. [69]. This tool includes *(i)* a text classifier using NLP features that analyses suspicious strings used by the application, *(ii)* a lightweight `smali` emulation technique to detect locking strategies, and *(iii)* a taint tracker for detecting file-encrypting flows. The system has then been further expanded by Zheng et al. [70] with the new name of `GreatEatlon`, and features significant speed improvements, a multiple-classifier system that combines the information extracted by text and taint-analysis, and so forth. However, the system is still computationally demanding, and it still strongly depends on a text classifier: the authors trained it on generic threatening phrases, similar to those that typically appear in ransomware or scareware. This strategy can be easily thwarted, for example, by employing string encryption [71]. Moreover, it strongly depends on the presence of a language dictionary for that specific ransomware campaign.
Yang et al. [75] have proposed a tool to monitor the activity of ransomware by dumping the system messages log, including stack traces. Sadly, no implementation has been released for public usage.

TABLE 4.1: An overview of the current state-of-the-art, ransomware-oriented approaches.

Work	Tool	Year	Static	Dynamic	Machine-Learning	Available
Chen et al. [62]	RansomProber	2018		✓		
Cimitile et al. [72]	Talos	2017	✓			
Gharib and Ghorbani [73]	Dna-Droid	2017	✓	✓	✓	
Song et al. [74]	/	2016		✓		
Zheng et al. [70]	GreatEatlon	2016	✓		✓	✓
Yang et al. [75]	/	2015		✓		
Andronio et al. [69]	HelDroid	2015	✓		✓	✓

Song et al. [74] have proposed a method that aims to discriminate between ransomware and goodware using process monitoring. In particular, they considered system-related features representing the I/O rate, as well as the CPU and memory usage. The system has been evaluated with only one ransomware sample developed by the authors, and no implementation is publicly available.
Cimitile et al. [72] have introduced an approach to detect ransomware that is based on formal methods (by using a tool called `Talos`), which help the analyst identify malicious sections in the app code. In particular, starting from the definition of payload behaviour, the authors manually formulated logic rules that were later applied to detect ransomware. Unfortunately, such a procedure can become extremely time-consuming, as an expert should manually express such rules.
Gharib and Ghorbani [73] have proposed Dna-Droid, a static and dynamic approach in which applications are first statically analysed, and then dynamically inspected if the first part of the analysis returned a suspicious result. The system uses Deep Learning to provide a classification label. The static part is based on textual and image classification, as well as on API calls and application permissions. The dynamic part relies on sequences of API calls that are compared to malicious sequences, which are related to malware families. This approach has the drawback that heavily obfuscated apps can escape the static filter; thus, avoiding being dynamically analysed.
Finally, Chen et al. [62] have proposed RansomProber, a purely dynamic ransomware detector that employs a set of rules to monitor different aspects of the app execution, such as the presence of encryption or anomalous layout structures. The attained results report a very high accuracy, but the system has not been publicly released yet (to the best of our knowledge).

Table 4.1 shows a comparison between the state-of-the-art methods for specifically detecting or analysing Android ransomware. It is possible to observe that there is a certain balance between static- and dynamic-based methods. Some of them also resort to Machine-Learning to perform classification. Notably, only HelDroid and GreatEatlon are currently publicly available.

4.2 On the Effectiveness of System API-related Information

As described in the previous section, machine learning has been increasingly used both by researchers and anti-malware companies to develop *ransomware-oriented* detectors. The reason for such a choice is that ransomware infections may lead to

permanent data loss, making their early detection critical. The main characteristic of such systems is that they rely on different types of information extracted from multiple parts of the apps (e.g. bytecode, manifest, native libraries, and so forth), which leads to using large amounts of features (even hundreds of thousands). While this approach is tempting and may seem to be effective against the majority of attacks in the wild, it features various limitations. First, it is unclear which features are essential (and needed) for classification, an aspect that worsens the overall explainability of the system (i.e. why the system makes mistakes and how to fix them). Second, increasing the types of features extends the degrees of freedom of a skilled attacker to perform targeted attacks against the learning algorithm. For example, it would be quite easy to mask a specific IP address if the attacker understood that this has a vital role in detection [17]. Finally, the computational complexity of such systems is enormous, which makes them unfeasible to be practically used in mobile devices, an important aspect to guarantee offline, early detection of these attacks.

In the original work by Maiorca et al. [76], the proposed detection method (called R-PackDroid) had allowed discriminating between ransomware, generic malware, and legitimate files by focusing on a small-sized feature set, i.e. System API packages. The idea of that work had been to overcome the limitations described above by showing that it was possible to solve a machine learning problem with a limited number of features of the same type. However, System API-based information does not only include *packages* but also *classes* and *methods* (particularly employed in other works, especially mixed with other feature types [15, 59]) that can — potentially — better define the behaviour of APIs. Intuitively, using finer-grained information leads to better accuracy and robustness in comparison to other approaches. Therefore, in the following, I present how we have explored such a possibility by progressively refining the System API-based information employed in the original, previous work [76]. In particular, we have inspected the capabilities of multiple types of System API-related information to discriminate ransomware from malware, and goodware, aiming to provide an answer to the following Research Questions:

- **RQ 1.** Does the use of finer-grained information related to System API (i.e. classes and methods) improve detection performances in comparison to more general System API packages?
- **RQ 2.** Is System API-based information suitable to detect novel attacks in the wild?
- **RQ 3.** Does using System API-based information provide comparable performances to other approaches that employ multiple feature types?
- **RQ 4.** Is System API-based information resilient against obfuscation attempts?

To answer such Research Questions, we have explored three types of System API-based information: the first one only uses information related to System API packages (as already shown in [76]), the second one analyses System API classes, and the third one employes information related to System API methods. We have evaluated the performances of the three systems on a wide range of ransomware, malware and goodware samples in the wild (including previously unseen data). Moreover, we have tested all systems against a dataset of ransomware samples that had been obfuscated with multiple techniques (including class encryption).

The attained results show that all System API-based techniques provide excellent accuracy at detecting ransomware and generic malware in the wild, by also showing capabilities of predicting novel attacks and resilience against obfuscation. More specifically, using finer-grained information even improves the accuracy at detecting previously unseen samples, and provides more reliability against obfuscation attempts. From a methodological perspective, such results demonstrate that it is possible to develop accurate systems by strongly reducing the complexity of the examined information and by selecting feature types that represent how ransomware attacks behave.

Finally, to demonstrate the practical suitability of System API-based approaches on Android devices, we have ported to Android R-PackDroid (the package-based strategy originally proposed in [76] and further explored in this work). Our application, which can detect both ransomware and generic malware in the wild, shows that methodologies based on System API can be implemented with good computational performances — e.g. with a processing timing that is acceptable for end users — even in old phones, and its a demonstration of a full working prototype being deployed on real analysis environments. R-PackDroid can be downloaded for free from the Google Play Store[1].

With this work, we have pointed out that it is possible to create effective, deployable, and reasonably secure approaches for ransomware detection by only using specific feature types. Hence, this work acts as the baseline to make use of explainability in order to make detection even more accurate and robust.

The organisation of the content of this work is the following: Section 4.2.1 describes the employed detection methods. Section 4.2.2 illustrates the experimental results attained with all of them, as well as a comparison between our systems and other approaches in the wild. Section 4.2.3 describes the implementation details of R-PackDroid and its computational performances. Section 4.2.4 discusses the limitations of the work, which is finally concluded by Section 4.2.5.

4.2.1 Method

As anticipated, the intuition for developing a ransomware-oriented detector is to rely on a smaller set of information (System API calls) that is typically employed in ransomware. However, as System APIs are also widely used in generic malware and legitimate files, this information type also allows detecting other attacks that differ from ransomware. In this way, it is possible to create a powerful, wide-spectrum detector that features a much lower complexity in comparison to other approaches. Accordingly, such a system may take as input an Android application, analyse it and return three possible outputs: *ransomware*, *generic malware*, or *trusted*. The analysis is performed in three steps:

- **Pre-Processing:** in this phase, the application is analysed to extract its `dexcode`. The required information is extracted by only inspecting the executable code and does not perform any analysis on other elements, such as the application Manifest. Only specific lines of code, which will be described later in this section, will be sent to the next module.

[1] `http://pralab.diee.unica.it/en/RPackDroid`

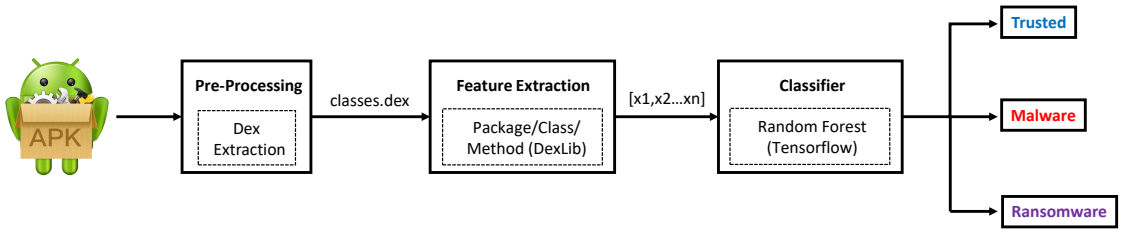


FIGURE 4.1: General Structure of a System API-based, ransomware-oriented system.

– **Feature extraction:** in this phase, the code lines received from the previous phase are further analysed to extract the related *System API* information (either packages, classes, or methods). The *occurrence* of such pieces of information is then counted; thus, producing a vector of numbers (*feature vector*) that is sent to a classifier.
– **Classification:** it is carried out through a *supervised* approach, in which the system is trained with samples whose label (i.e. benign, generic malware or ransomware) is known. Such a technique has been used in previous work with excellent results [15, 17, 59]. In particular, our approaches employ Random Forest classifiers, which are especially useful to handle multi-class problems, and which are widely used for malware detection. The complexity of such classifiers depends on the number of trees that compose them. Such a number must be optimised during the training phase.

The structure above is graphically represented in Figure 4.1. In the following, we provide more details about each phase of the analysis, by focusing in particular on the type of features that can be extracted from the application.

4.2.1.1 Preprocessing and Feature Extraction

The general idea of the first two phases is performing static analysis of the `Dalvik` bytecode contained in the `classes.dex` file. The goal is retrieving the *System API information* employed by the executable code of the application. The choice of System API information is related to two basic ideas:

– **Coherence with actions**: most ransomware writers resort to System APIs to carry out memory- or kernel-related actions (for example, file encryption or memory management). Focusing on user-implemented APIs (as it happens, for example, with Drebin [15]) exposes the system to a risk of being evaded by simply employing different packages to perform actions;
– **Independence from Training**: system API calls are features independent of the training data that are used. As a consequence, it is less likely that applications are not correctly analysed because they employ never-seen-before APIs only;
– **Resilience against obfuscation**: using heavy obfuscation routines typically leads to injecting system API-based code in the executable, which can be extracted and analysed, allowing to detect suspicious files.

Pre-processing is hence easily performed by directly extracting the `classes.dex`

file from the `APK`. Since `.apk` files are essentially zipped archives, such an operation is rather straight-forward.

Once pre-processing is complete, the `classes.dex` file is further analysed by the feature extraction module, which inspects the executable file for all `invoke`-type instructions (i.e. all instructions related to invocations) contained in the `classes.dex` code. Then, each invocation is inspected to extract the relevant API information for each methodology, according to a System API reference list that depends on the operating system version (in our case, Android Nougat — API level 25). Only the API elements that belong to the reference list are analysed. In this work, we consider three different methodologies, based on, respectively, package, class, and method extraction. If a specific API element is found, its occurrence value is increased by one.

In the following, we provide a more detailed description of the methodologies employed in this work by referring to the example reported in Listing 4.1. Such a listing comes from the `.smali` output, which uses a '/' as separator between package, class and method names. The code is parsed in three ways, according to each feature extraction strategy. For each example, we used a very small subset of the employed reference API.

Package Extraction We extract the occurrences of the System API packages (a total of 270 reference features), in the same way of the original work [76]. In the example of Listing 4.1, we use a subset composed of three reference API packages: `java.io`, `java.crypto` and `java.lang`. The four `invoke` instructions are related to the `javax.crypto` and `java.io` packages, which are counted respectively twice. The `java.lang` package is never used in this snippet. Hence, its value is zero.

Classes Extraction In this case, we extract the occurrences of the System API classes (a total of 4609 reference features). Notably, such classes belong to the System API packages of the previous methodology (and, for this reason, their number is significantly higher than packages). In the example of Listing 4.1, we use a subset composed of two reference API classes: `java.io.FileInputStream` and `javax.crypto.CypherOutputStream`, each of them appearing twice.

Methods Extraction In this third case, we extract the occurrences of the System API methods (a total of 36148 reference features). These methods belong to the System API classes of the previous methodology, leading to a very consistent number of features. This strategy is very similar to other ones employed by other systems (e.g.. [15, 59]), which have used these features together with user-implemented APIs and other features. In the example of Listing 4.1, we use a subset composed of four reference API methods: `read()` and `close()` from `java.io.FileInputStream`, `flush()` and `close()` from `javax.crypto.CypherOutputStream`. Each API call appears only once. Note that, although there are two methods named 'close', they belong to two different classes, and they are therefore considered as different methods.

4.2.2 Experimental Evaluation

In this section, I report the experimental results attained from the evaluation of the three API-based strategies. Note that, for the sake of simplicity and speed, we did

```
1
2 Code
3
4 Ljava/io/FileInputStream;->read([B)I
5 move-result v0
6 const/4 v5, -0x1
7 if-ne v0, v5, :cond_0
8 invoke-virtual {v1}, Ljavax/crypto/CipherOutputStream;->flush()V
9 invoke-virtual {v1}, Ljavax/crypto/CipherOutputStream;->close()V
10 invoke-virtual {v3}, Ljava/io/FileInputStream;->close()V
11
12 Feat. Vectors
13
14 Packages - [2 2 0]
15
16 Classes - [2 2]
17
18 Methods - [1 1 1 1]
```

LISTING 4.1: An example of feature extraction by considering a small number of reference features.

not run the experiments on Android phones, but on an X86 machine. However, we built a full, working implementation of one of the three approaches, which can be downloaded from the Google Play Store (see next section).

The rest of this section is organised as follows: I start by providing an overview of the dataset employed in our experiments. Then, I describe the results attained by four evaluations. The first one aimed to establish the general performances of API-based approaches by considering random distributions of training and test samples. The second one aimed to show how API-based approaches behaved when analysing samples released after the training data. The third one aimed to show a comparison between our API-based approaches and other systems that employed mixed features. Finally, we evaluate the resilience of API-based approaches against obfuscation techniques and evasion attacks.

4.2.2.1 Dataset

In the following, I describe the dataset employed in our experiments, including those in other works of this thesis. Without considering obfuscated applications (which are going to be discussed in Section 4.2.2.3), we have obtained and analysed 39157 apps, which are organised in the three categories mentioned in Section 4.2.1.

Ransomware The 3017 samples used for our ransomware dataset have been retrieved from the VirusTotal service[2] (which aggregates the detection of multiple anti-malware solutions) and from the HelDroid dataset[3] [69]. With respect to the samples obtained from VirusTotal, we have used the following procedure to obtain the samples: *(i)* we have searched and downloaded the Android samples whose anti-malware label included the word *ransom*; *(ii)* for each downloaded sample, we have extracted its family by using the AVClass tool [77], which essentially combines the various labels provided by anti-malware solutions to create a unique label that identifies the sample itself; *(iii)* we have considered only those samples whose family was coherent to ransomware behaviours, or was known to belong to ransomware.

[2] http://www.virustotal.com
[3] https://github.com/necst/heldroid

TABLE 4.2: Ransomware families included in the employed dataset.

Family	Samples
Locker	752
Koler	601
Svpeng	364
SLocker	281
Simplocker	201
LockScreen	122
Fusob	120
Lockerpin	120
Congur	90
Jisut	86
Other	280

In general, our goal has been obtaining a representative corpus of ransomware to ascertain the prediction capabilities of API-based techniques. For this reason, the dataset includes families that perform both device locking (such as `Svpeng` and `LockScreen`) and encryption (such as `Koler` and `SLocker`).

Malware and Trusted We consider a dataset composed of 17744 Android malware samples that do not belong to the ransomware category, taken from the following sources: *(i)* Drebin dataset, one of the most recent, publicly available datasets of malicious Android applications[4] (which also contains the samples from the Genome dataset [78]); *(ii)* `Contagio`, a popular free source of malware for X86 and mobile; *(iii)* VirusTotal. These samples have been chosen to verify whether even non-ransomware attacks could be detected with features that are particularly effective at classifying ransomware samples.

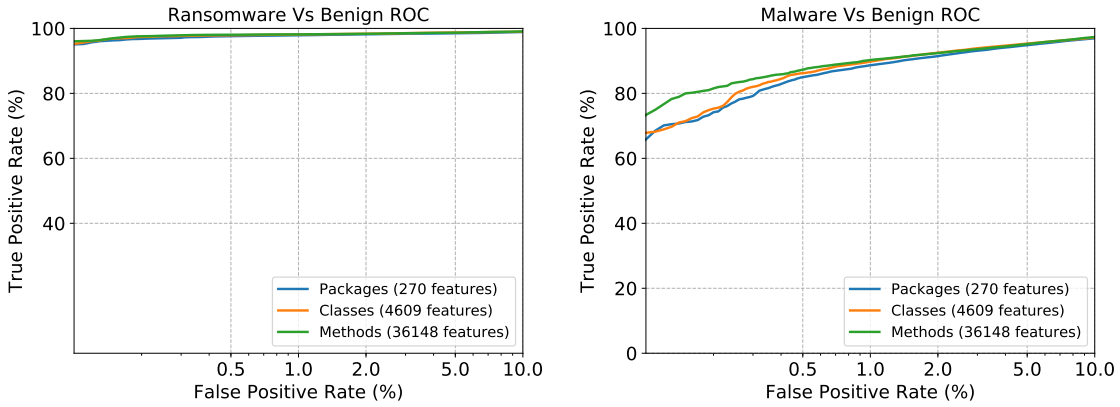
In order to download trusted applications, we have resorted to two data sources: *(i)* we have crawled the Google Play market using an open-source crawler[5]; *(ii)* we have extracted a number of applications from the AndroZoo dataset [79], which features a snapshot of the Google Play Store, allowing to access applications without crawling the Google services easily. We have obtained 18396 applications that belong to all the different categories available on the market. We have chosen to focus on the most popular apps to increase the probability of downloading malware-free apps.

4.2.2.2 Experiment 1: General Performances

In this experiment, we evaluate the general performances of System API-Based methods (described in Section 4.2.1) at detecting ransomware and generic malware. To do so, for each strategy, we randomly split our dataset by 50%; thus, using the first half to train the system and the second half to evaluate the system. The number of trees of the random forest was evaluated by performing a 10-fold cross-validation on the training data. We have repeated the whole process 5 times, and we have averaged the results by also determining the standard deviation.

[4] `https://www.sec.cs.tu-bs.de/~danarp/drebin/`

[5] `https://github.com/liato/android-market-API-py`



(a) (b)

FIGURE 4.2: Results from Experiment 1. The ROC curves (averaged on 5 splits) attained by the three System API methods for ransomware (a) and generic malware (b) against benign samples are reported.

Considering the multi-class nature of the problem, we represent the results by calculating the ROC curve for each API-based strategy in two different cases:

- **Ransomware against benign samples:** the crucial goal of this work has been detecting ransomware attacks and, more importantly, *avoiding them from being considered benign files*. A critical mistake would most likely compromise the whole device by locking it or encrypting its data. For this reason, it is essential to verify whether ransomware attacks can be confused with benign samples;
- **Generic malware against benign samples:** even if System API-based strategies are employed to detect ransomware, they could also be used to classify generic malware (see Section 4.2.1). Hence, the goal here is to verify, from a practical perspective, if System API-based information can correctly detect other non-ransomware attacks and distinguish them from legitimate files.

Figure 4.2 shows the ROC curves that describe the performances attained on ransomware and generic malware detection by the three System API-based methods (packages, classes, methods). By observing these curves, we can deduce the following facts:

1. All System API-based techniques were able to precisely detect more than 97% of ransomware samples, with only 1% of false positives. Because our dataset included a consistent variety of families, we claim that all strategies can detect the majority of ransomware families in the wild. Worth noting, there are no differences in results between using packages, classes, or methods. This result means that, concerning general performances, using finer-grained features does not improve detection.

2. All System API methods featured good accuracy with relatively low false positives (around 90% at 1%, more than 95% at 2%) at detecting generic malware. While using class-related features did not bring significant improvements to

detection, using methods allowed for a 10% improvement for false-positive values inferior to 0.5%.

To better understand the results attained by our strategies, Figure 4.3 reports a ranking of features used by the classifier for each strategy (respectively, packages, methods, and classes), according to their discriminant power. The feature ranking is calculated according to the features Information Gain IG, given by the following formulation:

$$IG(T, a) = H(T) - H(T|a) \tag{4.1}$$

where $H(T)$ is the overall entropy for the whole dataset T and $H(T|a)$ is the average entropy obtained by splitting the set T using the attribute a. The higher is the gain, the more relevant the feature is. As a result, note how the information gain for each feature is not so high, meaning that the system does not particularly overfit on specific information and that the final decision is taken by considering a combination of multiple features. At the same time, each feature value is reduced, in comparison to packages, by one magnitude for classes and methods. In other words, using more features allows for distributing the importance of the analysed information through more elements. This characteristic is two-faced: while it makes the overall behaviour of the system less interpretable, it may increase the effort that an attacker has to make to evade the system.

Analysing the most discriminant methods can give a clearer idea of which information is used to classify applications. Features are related to string building (e.g. the `ToString` method), Array management (e.g. `ArrayList@size`, `ArrayList@remove`), creation of folders (e.g. `File@mkdirs`), SMS, URI, layout management, and so forth. These features may be easily associated with both ransomware and malware behaviour, and the same behaviour is shown on classes and packages.
A careful examination of the feature ranking may also help to understand why specific samples are regarded as false positives or false negatives. In particular, detection is performed by weighting the information provided by a combination of the most discriminant features. For this reason, in some cases, specific ransomware (or generic malware) samples may contain discriminant features that are distributed differently to the malicious training distribution. For example, the `toString` call may appear, on average, 5 times on ransomware, but one specific sample may feature it only 1 time. This phenomenon may occur for various reasons, including the possibility that an attacker may be using customized variants of System API information to avoid detection (see Section 4.2.4). Likewise, the techniques used to create the sample (e.g. repackaging) may have an impact on the distribution of the features. Further refinement of the feature list (or a change of the weights of a specific feature) may help to reduce the amount of misclassified samples.

4.2.2.3 Experiment 2: Temporal Performances

In this experiment, we assess the capabilities of System API-based methods at detecting ransomware samples that were first seen (according to the creation date of the `classes.dex` executable belonging to each application) *after* the data that were used for the training set. This assessment is useful to understand if, without constant upgrades to its training set, such methods would be able to detect novel, unseen ransomware samples.

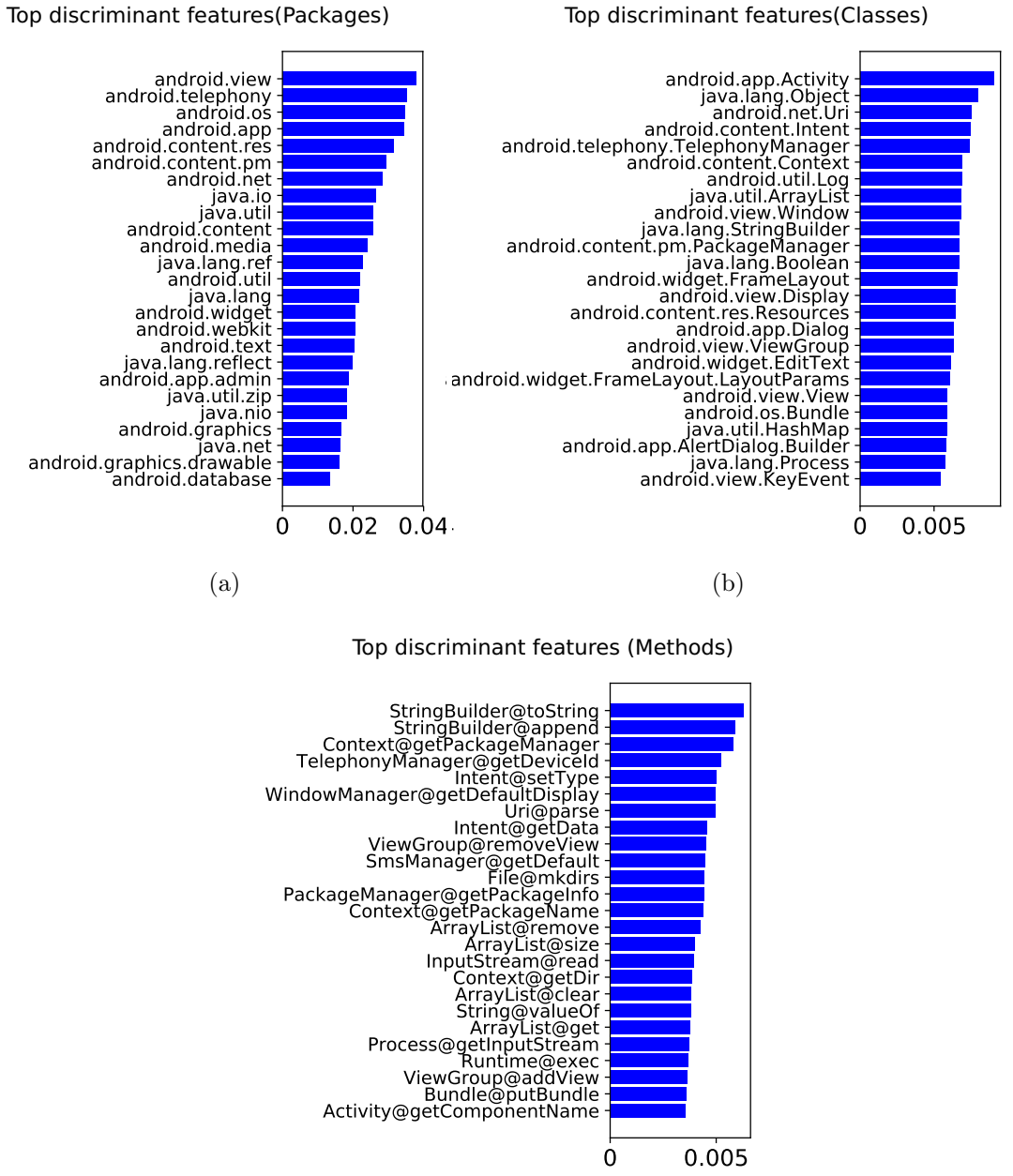


FIGURE 4.3: Results from Experiment 1. We report the top-25 features, ordered by the classifier information gain (calculated by averaging, for each feature, the gains that were obtained by training the 5 splits), for each methodology: (a) for packages, (b) for classes, (c) for methods.

For this assessment, we have included in the training set only the samples that were first seen before a date D_{tr}, and we have tested our system on a number of ransomware samples that had been released on a date D_{te} for which $D_{te} > D_{tr}$ (we choose the ROC operating point of the system corresponding to a false positives value of 1%). We have performed our tests by choosing different values of D_{te}, where D_{tr} is December 31st, 2016. Concerning test data, we point out that the samples (which were extracted by the VirusTotal service) are unevenly distributed through the months. More specifically, the number of ransomware samples that had been submitted to the VirusTotal service was significantly different for each month of 2017. We have been able to retrieve only a little amount of samples whose first release date was between January and September 2017. Conversely, we could obtain a consistent amount of samples whose D_{te} was October and November 2017. Therefore, we have grouped the samples gathered in subsequent months to obtain temporal ranges with similar amounts of testing data. We considered three main ranges for D_{te}: *(i)* January to September 2017; *(ii)* October 2017; *(iii)* November 2017.

FIGURE 4.4: Results of the temporal evaluation for the System API-based strategies. The accuracy values are reported for the three System API-based detection. The training data belong to 2016, while the test data is composed of ransomware released in different months of 2017.

Results are provided in Figure 4.4, which shows that by training the system with data retrieved in 2016, class- and method-based strategies can accurately detect ransomware test samples released in 2017. However, the package-based strategy struggles at detecting the test-set from November 2017. Notably, in comparison with class- and method-based strategies, the package-based approach shows almost a 10% accuracy loss when analysing samples released till October, and more than 20% accuracy loss for samples released in November. Conversely, the other two methods exhibit stable accuracy on each temporal range. This result is particularly interesting, as it shows that the prediction of novel samples can be significantly improved by employing finer-grained features. However, the results attained by package-based

features are nevertheless encouraging, as they showed that even a reduced number of features could attain good performances at detecting novel attacks. Overall, this experiment has further confirmed that System API-based strategies can predict new ransomware attacks with good accuracy, even on test data released. In this case, using finer-grained features brings a consistent advantage to detection.

4.2.2.4 Experiment 3: Comparison with Other Approaches

This section proposes a comparison between System API-based strategies and other state-of-the-art approaches. We were particularly interested in comparing our approach to other publicly available ones, with a special focus on those who were specifically designed to detect ransomware. Additionally, we have considered those approaches that, albeit not explicitly designed to detect ransomware, could tell if the analysed sample is malicious or not. In particular, almost all of the analysed tools (except for Talos [72], which does not employ machine learning) discriminate between two classes (malware and benign or ransomware and benign), while our approaches discriminate between three classes (ransomware, malware, and benign). Hence, it is interesting to observe how increasing the number of classes may impact the precision of the analysis.

We have performed a temporal comparison of all systems on the ransomware samples released in 2017 (for a total of 512 samples) by using (when possible) all data released until 2016 as training.

The state-of-the-art approach that is closest to what we proposed in this work (while being publicly available[6]) is GreatEatlon [70]. Notably, it was not possible for us to control the trained model of the system (it was only possible to choose among a restricted set of classifiers), or to train it with new data. Nevertheless, the system was released in 2016, meaning that data that was first seen in 2017 was for sure not included in its training set. Although not specifically tailored to ransomware detection, we have also tested the performances of RevealDroid (which is publicly available[7] [59]) on the same test data. In this case, we could train the system with the same data used in our systems, which has allowed us to provide a fairer comparison. Finally, we have also tested the performances of the Android version of IntelliAV (available on the Google Play Store) [80]. As in GreatEatlon, we could not control the training data of the system. Moreover, as IntelliAV reports three levels of risk for each app (safe, suspicious, risky), we considered as malicious also the files that were labelled as suspicious by the system.

As classifier for GreatEatlon we have chosen Stochastic Gradient Descent (SGD), since this was the classifier that best performed on our test samples. Concerning RevealDroid, we have chosen the linear SVM classifier, as this was the one that provided the best results in the original work [59]. IntelliAV only employes Random Forests. Results are reported in Table 4.3.

The attained results show that System API-based techniques obtain very similar performance to RevealDroid (which could only, however, classify samples either as malware or benign). Such results are particularly interesting if we consider that RevealDroid extracts a huge number of features (more than $700,000$) from multiple characteristics of the file, including native calls, permissions, executable code

[6] `https://github.com/necst/heldroid`

[7] `https://seal.ics.uci.edu/projects/revealdroid/`

TABLE 4.3: Detection performances for System API-based strategies, GreatEatlon, IntelliAV, RevealDroid, and Talos on 512 ransomware test files released in 2017, by using training data from 2016. We use the ND (Not Defined) to indicate that a certain tool cannot provide the corresponding label for the analysed samples.

System	Benign	Generic Malware	Ransomware
Talos	3	0	509
System API (Methods)	7	12	493
System API (Classes)	10	15	487
System API (Packages)	11	32	469
GreatEatlon	118	*ND*	394
RevealDroid	0	512	*ND*
IntelliAV	18	494	*ND*

analysis, which also depends on the training data. With a much simpler set of information, we are able to obtain very similar performance concerning accuracy. This result is especially interesting from the perspective of adversarial attacks, as using fewer features for classification can make the system more robust against them (the attacker can manipulate less information to evade the system) [81]. The performance attained by System API-based approaches is also better than IntelliAV, which employes a combination of different features (including permissions, user-defined API, and more). System API-based strategies also perform significantly better than GreatEatlon, which based its detection also on information extracted from strings and language properties. Notably, using methods significantly improves the accuracy performances in comparison to packages and classes, in line with what was obtained from Experiment 2. Finally, we have analysed the performance attained by Talos [72][8], a static analysis tool that employs logic rules to perform detection (hence, without machine learning). The attained results are very encouraging, but they strongly depended on the set of rules that have been (manually) established to perform ransomware detection. Conversely, System API-based approaches do not require any manual definition of the detection criteria. Additionally, the analysis times of Talos are significantly slower than the ones attained by methods proposed in this work (an average of 100 seconds per application, 400 times slower than ours) — see Section 4.2.3.1.

4.2.2.5 Experiment 4: Resilience against Obfuscation

The goal of this experiment is to assess the robustness of System API-based strategies against obfuscated samples, i.e. understanding whether the application of commercial tools to samples can influence the detection capability of the systems. This evaluation is important, as commercial obfuscation tools are quite popular nowadays since they introduce good protection layers against static analysis (e.g. to avoid pieces of legitimate applications to be copied). Previous work has shown that attackers could exploit this aspect by obfuscating malware samples with such tools; thus, managing to bypass anti-malware detection [71].

In this experiment, we primarily focus on obfuscated samples whose original (i.e.

[8]We have obtained Talos directly from the authors, as it is not currently publicly available.

non-obfuscated) variant was already included in the training set. Such a choice has been made because we wanted to assess if obfuscation was enough to influence the key-features of System API-based methods; thus, *changing the classifiers' decision for a sample whose original label was malicious*.

To this end, we have employed a test-bench of ransomware obfuscated with the tool DexProtector[9], a popular, commercial obfuscation suite that allows for protecting Android applications through heavy code obfuscation. Although such a tool is mostly used for legitimate purposes (e.g. protection of intellectual properties), it can also be used by attackers to make malicious applications harder to be detected. Out of the 3017 ransomware samples, we could obfuscate 2668 samples (the remaining could not be obfuscated due to errors of the obfuscation software) with three different strategies (for a total of 8004 obfuscated samples). The strategies employed to obfuscate samples are the following:

- **String Encryption:** this strategy encrypts strings that are identified by `const-string` instructions, and injects a user-implemented method that performs decryption at runtime;
- **Resource Encryption:** it encrypts the external resources contained in the `res` and `assets` folders. To do so, it adds System API information to the `classes.dex` file, in order to properly manage the encryption routines;
- **Class Encryption:** this strategy encrypts user-implemented classes, and injects routines that allow performing dynamic loading of such classes.

Figure 4.5 reports the accuracy attained by the three System API-based strategies against the obfuscated samples. Such results show that all the detection strategies (without significant differences between each other) are resilient against obfuscation attempts. However, Class Encryption deserves separate consideration. This strategy employs heavy obfuscation, and it was explicitly performed to defeat static analysis. Typically, none of the static-based techniques that analyse the executable file should be able to detect such attacks correctly. However, this obfuscation strategy introduces a very regular sequence of System API-based routines that manage runtime decryption of the executable contents.

For this reason, it is sufficient to inject only one sample inside the training set to make all obfuscated samples to be detectable. Hence, we added the +1 mark to Class Encryption. Notably, this may create false positives when legitimate samples are obfuscated with the same strategy. Nevertheless, it is sporadic to find such applications, as class encryption strongly decreases the application performances [71], and much simpler obfuscation techniques are generally used.

4.2.3 Implementation and Performances

Although many solutions have been proposed in the wild to detect ransomware and generic malware, almost none (with the exception, for example, of [80]) was ported to Android devices, often due to the complexity of the proposed approaches. However, an offline, on-device solution is very useful to perform early detection of applications downloaded, for example, from third-party markets (which are more subjected to malware attacks). For this reason, and also to demonstrate the suitability of System API-based approaches, we have ported the simplest of the three

[9] `https://dexprotector.com/`

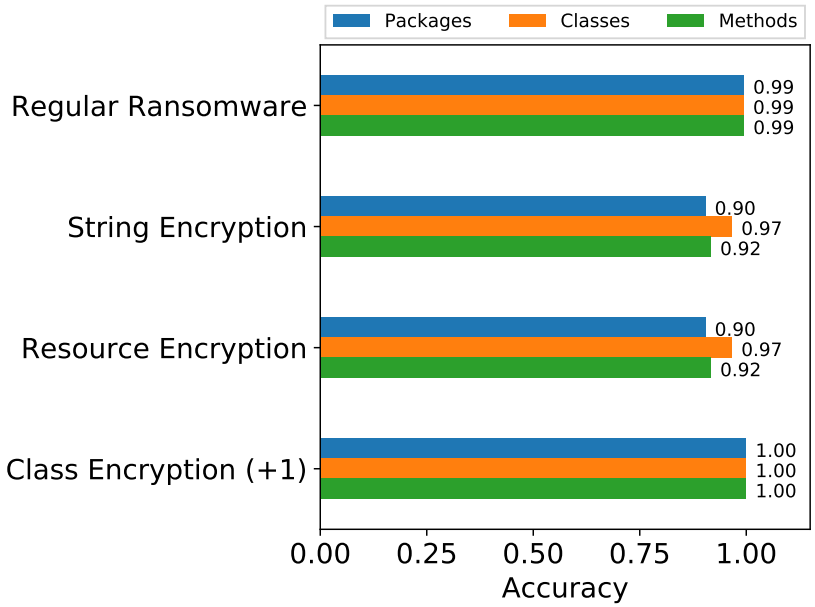


FIGURE 4.5: Accuracy performances attained on ransomware samples that have been obfuscated with three different techniques. The accuracy values are reported for the three System API-based detection.

strategies (Package-based) with the name of R-PackDroid (as it implements the same approach introduced in our previous work [76]). This implementation scans for any downloaded, installed, and updated applications, and it classifies them as ransomware, malware, or legitimate. If an application is found malicious, the user can immediately remove it.

R-PackDroid has been designed to work on the largest amount of devices possible. Hence, during its development, we have been focusing on optimizing its speed and battery consumption. For this reason, we have avoided any textual parsing of bytecode lines (which can be attained by transforming the `.dex` file to multiple `.smali` files with ApkTool). Therefore, we have resorted to DexLib, a powerful parsing library part of the baksmali[10] disassembler (and used by ApkTool itself), to directly extract method calls and their related packages. This library has allowed obtaining a very high precision at analysing method calls and significantly reduces the presence of bugs or wrong textual parsing in the analysis phase.

The classification model has been implemented by using Tensorflow[11], an open source, machine learning framework that has been designed to be also used in mobile phones. In particular, we have adapted its Random Forest implementation (`TensorForest`) to the Android operating system. Notably, our Android application only performs classification by using a previously trained classifier. The training phase is carried out separately on standard X86 architectures. This choice was made to ensure the maximum easiness of use to the final user; thus, reducing the risk of invalidating the existing model.

Figure 4.6 shows an example of the main screen of R-PackDroid. The application is parsed either when it is downloaded from any store, or when the user decides to

[10] `https://github.com/JesusFreke/smali`
[11] `https://www.tensorflow.org/`

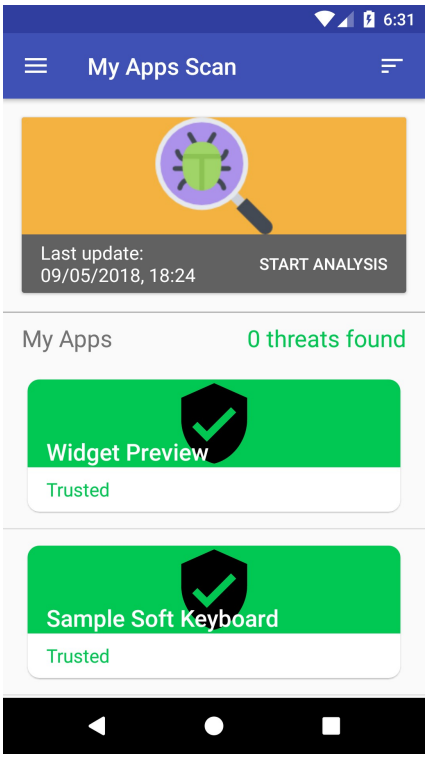


FIGURE 4.6: An example of the Android R-PackDroid screen.

scan it (or to scan the whole file system). Each application is identified by a box, whose colour is associated with the application label (green for trusted, red for malware, and violet for ransomware). After getting the result, by clicking on each box related to the scanned application, it is possible to read more details about the packages that it employs, as well as general information such as the app hash and size. Moreover, if the user believes that the result reported by R-PackDroid is wrong, she can report it by simply tapping a button (a privacy policy to accept is also included). To this scope, we resort to the popular service FireBase. [12] R-PackDroid is available for free on the Google Play Store. [13]

4.2.3.1 Computational Performances

We have analysed the computational performance of R-PackDroid by running it both on X86 and Android environments. In particular, we have focused on extracting the time interval between the `APK` loading and the generation of the feature vector for 100 benign samples (grouped by their `APK` size)[14]. The choice of benign samples has been made because they are typically more complex to be analysed in comparison with generic malware and ransomware. We have firstly run our experiments on a 24-core Xeon machine with 64 GB of RAM. The attained results, shown in Figure 4.7, prove that our system could analyse even huge applications in less than 0.2 seconds.

To evaluate the performance of R-PackDroid on a real Android phone, we have

[12] `https://firebase.google.com/`

[13] As the app is not currently mantained, Android versions until 7.1 are supported).

[14] The elapsed time to classify a sample, i.e. to read its feature vector and get the final label, is negligible.

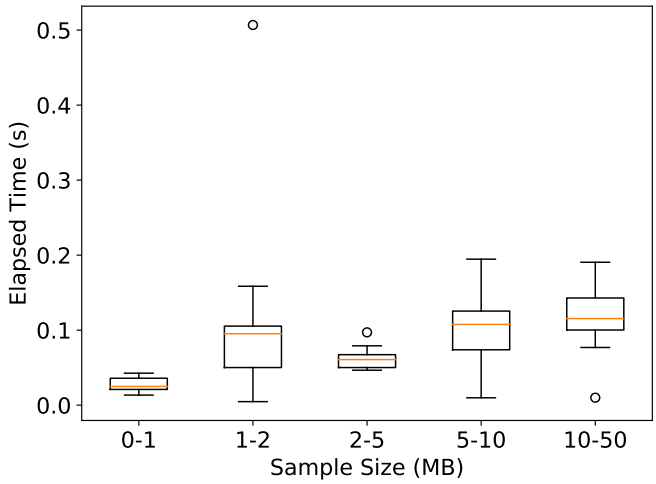


FIGURE 4.7: Analysis performances on a X86 workstation, with the elapsed time in seconds, for different `APK` sizes.

run the same analysis on a Nexus 5, a 5-year-old[15], 4-core device with 2 GB of RAM, equipped with the 6.0.1 version of Android. Results are reported in Figure 4.8. Even if the analysis times are slower than X86 machines, and even if we have been using, in this case, the slowest version of the algorithm, the average analysis time for very large apps is slightly more than 4 seconds. This result — at the time of the test — was very encouraging, and it showed that R-PackDroid could be safely used even on old phones. The higher dispersion of the time values, in comparison to the ones attained in the previous picture, was possibly caused by the presence of other background processes in the device.

Finally, it is also important to observe that the analysis time is not strictly proportional to the `APK` size, as the file may contain additional resources (e.g. images) that increase the `APK` size, without influencing the size of the `dexcode` itself. For this reason, it was not surprising to see the attained average values do not necessarily increase with the `APK` size.

4.2.4 Discussion and Limitations

The results attained in sections 4.2.2 and 4.2.3 can be summarised with the following findings:

- **Finding 1:** system API-based information can be effectively used — *alone* — to properly distinguish ransomware from generic malware and legitimate applications;
- **Finding 2:** using finer-grained information (classes and methods), albeit involving more features in the analysis, has brought significant improvements to accuracy when detecting previously-unseen samples. Moreover, using API methods enables having more accuracy under low false positives values;
- **Finding 3:** system API-based approaches can obtain comparable performances to other approaches that involve more features of different types;

[15] At the moment of the test, end of 2018

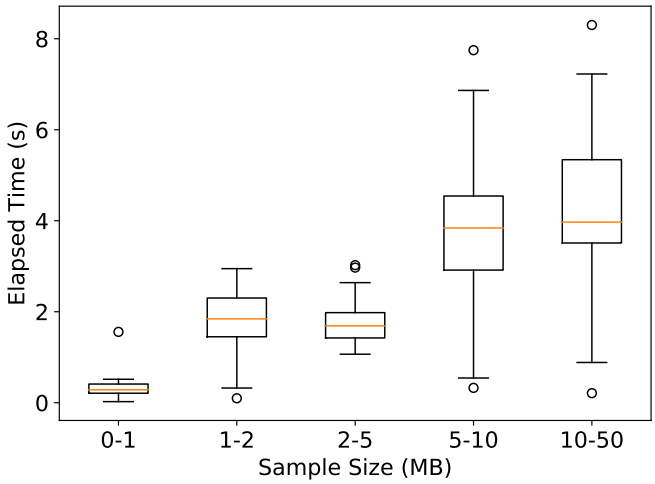


FIGURE 4.8: Performance analysis on a real device, with the elapsed time in seconds, for different `APK` sizes.

- **Finding 4:** system API-based approaches guarantee robustness against typical obfuscation strategies such as string encryption. However, by including a few obfuscated samples in the training set, it could also be possible to detect heavy, anti-static obfuscation techniques such as class encryption;
- **Finding 5:** system API-based approaches are well suitable to be ported and implemented on mobile devices, with excellent computational performances even on very large applications.

We point out that it would be possible to evade the proposed System API-based approaches by replacing a part of the System-related packages, classes, or methods with semantically equivalent, user-implemented ones. For example, attackers may have two possibilities to replace System API-based methods: *(i)* creating copies of the original instructions of the methods and injecting them into fake methods; *(ii)* re-implementing the methods by using customised instructions/logic. However, these two approaches may feature some critical limitations. In the first approach, the attacker is forced to import the copies of the instructions to the `dex` code (as the methods become user-implemented). However, the imported codes may contain further references to other System API-based methods, which would need to be replaced. Therefore, this procedure may become unfeasible, considering the wide variety of calls that can be invoked. The second approach may be hard to implement if the methods to be replaced are very sophisticated (e.g. methods related to cryptography or the execution of activities). Additionally, functionalities needed by the attacker may only be accessible through the System APIs.

There would also be the possibility that a skilled attacker attempts to evade System API-based detection algorithms by performing adversarial machine learning attacks, such as test-time evasion [17, 27]. In this scenario, the goal would be evading the classifier detection with a minimal number of changes by performing fine-grained modifications to the features of the analysed test samples. However, this strategy may be challenging to be performed in practice. The problem, also known in the literature as *Inverse Feature Mapping* [20, 27, 82], is constructing the real sample that implements the modifications made to the feature vectors. As the changes to the feature vector would involve the injection or removal of specific

System API-based information, they may not be feasible. I cover and inspect these adversarial-related aspects as well as the practical creation of evasive samples in Chapter 5.

As highlighted in Chapter 3, it is also worth noting that since Android Oreo, Google introduced new defences against background processes that are typical of ransomware (e.g. the ones that directly lock the device). However, this does not exclude other malicious actions on the application level. For this reason, it is always better to have an additional system that can detect attempts at performing malicious actions.

Finally, we also point out that, during our tests, we found samples that could not be analysed due to crashes and bugs of the DexLib library, and that have therefore been excluded from our analysis. However, their percentage (regarding the whole corpus that we have analysed) is negligible (less than 1% of the whole file corpus).

4.2.5 Conclusions

With this work, we have provided a detailed insight into how System API-based information could be effectively used (also on a real device) to detect ransomware and to distinguish it from legitimate samples and generic malware. The attained experimental results have demonstrated that, by using a compact set of information tailored to the detection of a specific malware family (ransomware), it is possible to achieve detection performance (also on other malware families) that is comparable to systems that employed a much more complex variety of information. Moreover, System API-based information has also proved to be valuable to detect obfuscated samples that focused on hiding user-implemented information. Notably, although it is tempting to combine as many information types as possible to detect attacks (or to develop computationally heavy approaches), it may not be the only feasible way to construct accurate, reliable malware detectors. For this reason, we claim that future work should focus on developing reliable, small sets of highly discriminant features that cannot easily be manipulated by attackers (with a particular reference to machine learning attacks). Moreover, a clear understanding of the impact of each feature on the classifier detection can help analysts understand *(i)* the classifiers' errors, so to improve their detection capabilities. *(ii)* the attacks' behaviour, so to identify spurious patterns. The thesis proposes to cover the latter aspect in the section that follows.

4.3 Explanation-driven Ransomware Characterisation

As illustrated in Section 4.1, the typical problem formulation for Android malware detectors is strictly correlated to the use of a wide variety of features covering different characteristics of the entities to classify. Moreover, Section 4.2 has shown that ransomware developers typically build such dangerous apps so that normally-legitimate components and functionalities (e.g. encryption) perform malicious behaviour; thus, making them harder to be distinguished from genuine applications. Differently from the goals of the previous work (Section 4.2), with this one by Scalas et al. [68] we have investigated if and to what extent state-of-the-art explainabil-

ity techniques help to identify the features that characterise ransomware samples, i.e. the properties that are required to be present in order to combat ransomware offensives effectively. In this regard, our contribution is threefold:

1. We present a first approach that enables a designer to select the explainability technique that is most suitable to the goal mentioned above; our approach presumes to be agnostic with respect to the learning algorithms;

2. We propose practical strategies for identifying the features that characterise generic ransomware samples, specific families, and the evolution of such attacks over time.

3. We counter-check the effectiveness of our analysis by evaluating the prediction performance of classifiers trained with the discovered relevant features.

In this way, we believe that our proposal can help cyber threat intelligence teams in the early detection of new ransomware families, and, above all, could be a starting point to help designing other malware detection systems through the identification of their distinctive features. Our approach is presented in Section 4.3.1, starting from the strategy to perform ransomware detection and followed by the rationale behind the usage of explanation techniques for identifying relevant features. Since the methods we consider have been originally designed to indicate the most influential features for a single prediction, we propose to evaluate their output against multiple samples to understand the *average* role of each feature. Notably, tailoring such techniques to our domain and goal requires attentive checks. Therefore, in our experimental analysis (Section 4.3.2) we first verify their suitability, and we eventually analyse their output to extract information. I discuss the limitations of our approach, together with future research paths, in Section 4.3.3.

4.3.1 Ransomware Detection and Explanations

In this section, I present our method. More specifically, I illustrate the rationale behind the usual design process of Android ransomware detectors and the resulting features (Section 4.3.1.1); then, I describe our proposal for leveraging gradient-based techniques in the design phase (Section 4.3.1.2).

4.3.1.1 Detector Design

The method used for this work mostly follows the one described in Section 4.2. However, in this case, we aim at understanding if the features identified in such previous work and the literature, turn out to be truly characteristic of the ransomware behaviour, besides being effective for detection. Accordingly, we point this out through explainability techniques, but we also consider a different setting. Keeping a fully static analysis setting, we have chosen to start exploring our proposed approach making use of two types of features: *(i)* the occurrence of System API package calls and *(ii)* the request of permissions. The rationale behind this choice has been the will to understand how two different kinds of features relate when inspected through explanation techniques. Moreover, with such an approach, we could later evaluate other varieties of features proposed in the literature — e.g. dynamic analysis ones to address the typical limits of static analysis — and include them if they truly

identify ransomware actions.
Another change in the setting of this work with respect to Section 4.2 comes from considering a bi-class setting, i.e. a classification system that is able to discriminate legitimate samples from ransomware samples, without the capability of identifying generic malware. I discuss this choice later in Section 4.3.3.

Before illustrating how we employ explanations, I briefly recall the motivation for the usage of requested permissions as features: the Android operating system requires the app developer to request permission to use a particular functionality expressly. Depending on its dangerousness, that permission can be granted automatically or after explicit user agreement. This distinction is also explicitly stated in the Android platform documentation as *protection level*; therefore, it is straightforward to consider permissions as useful features. For example, we can expect that common permissions (e.g. `INTERNET`, `ACCESS_NETWORK_STATE`) will be associated with trusted apps, while *dangerous* ones, such as `WRITE_EXTERNAL_STORAGE`, could be typical of ransomware samples.

4.3.1.2 Explaining Android Ransomware

In the following, I illustrate our proposal on how to take advantage of *gradient-based* explanations (see Section 2.2.1.1), aiming to identify a unique, reliable, and coherent set of relevant features that is independent of the specific *(i)* model, *(ii)* explanation method, and *(iii)* dataset. Accordingly, I first describe the potential influence of these three elements and how we propose to address it. Then I illustrate our information extraction process.

Influencing Factors A first concern arises around the choice of the classifiers. Since several types of them can be used with success for this detection problem, *there could be as many different explanations as to the number of potential classifiers*. For this reason, it is necessary to verify if the specific learning algorithm affects the output of the attributions. Therefore, a reasonable check to perform is to compare the explanation vectors within a set of plausible classifiers, one explanation technique at a time. Complementary to this aspect, we should get insights about which of the three explanation techniques is the most *accurate*, or to what extent they are equivalent. In this way, if all the explanations are mostly similar, it is possible to consider only one of them. Therefore, for each model, we compare its attributions across the different attribution techniques, similar to the procedure performed by Warnecke et al. [83]. It is worth noting that this approach gives rise to the following tricky issue: *how can we guarantee that similar attributions imply faithful explanations?* In this regard, in our setting, all the techniques are gradient-based, which are known to be reasonably effective; hence, we make the assumption that all of them are suitable despite their peculiarities.
The third concern comes from the influence of the data on the explanations; specifically, the possibility that *attributions could not be associated with the data labelling, and, consequently, to what the model learns*. For example, if attributions do not change after random ransomware samples are assigned to the trusted class, then the model would be bounded by the samples themselves, rather than by what it learned through the training phase. This aspect can be of particular interest for Gradient*Input and Integrated Gradients, where the input is part of their computa-

tion. In this case, we follow the method proposed by Adebayo et al. [84], where the ordinary attributions of each classifier are compared to the ones of a correspondent classifier trained with randomised labels.
I cover all the above-mentioned aspects in Section 4.3.2.2, where we practically perform different comparisons through the use of correlation and similarity metrics, described in Section 4.3.2.1.

Knowledge Extraction The above concerns and our proposed answers for them allow us to approach the problem with more awareness about the caveats that these explainability techniques pose. Nevertheless, there are a few other aspects that are worth focusing on. As the first remark, both the types of features of our setting (API calls and permission requests) are examples of *sparse* features. More specifically, each application uses only a small amount of APIs and permissions; consequently, each sample exhibits a few non-zero (used) features. This fact marks a non-negligible difference with respect to other domains, such as image recognition, where the features (typically raw pixels) are all always used. Therefore, when using explanations to characterise the Android ransomware samples under test, we expect the attributions to be sparse as well. Moreover, as investigated by Lage et al. [85], sparsity is one of the factors that make explanations more *comprehensible* for the human expert. All other aspects being equal, we will then favour the techniques that satisfy this requirement the most.

Once a specific set of reliable attributions is established, we should find a strategy to analyse them and extract information concretely. In particular, looking at the predictions' relevance values, we aim to catch the *average* role of the features onto the samples' characterisation. Since the attribution methods we have chosen provide us with a unique explanation for each prediction, it is necessary to define the concept of average in practice. Figure 4.9 shows an example of the distribution of the attributions obtainable with our setting, which I will describe in Section 4.3.2. In this figure, two features representing two permissions —in this thesis, we indicate permissions with capital letters — are shown; they are calculated with Integrated Gradients against trusted samples for an MLP classifier.
The significant aspect to observe is that, in our setting, the distribution of relevance values is typically bimodal. The attribution values could be zero for some samples and be condensed around a certain value level for some others. In particular, they exhibit a positive value if the feature converges towards the ransomware class; negative otherwise. Therefore, we could choose a synthetic metric that expresses the central tendency of this kind of distribution. In our work, *we consider the median value*. In this way, we highlight a feature as relevant when, for most of the samples, it does not exhibit a zero value. Although using a unique, synthetic measure for describing the attributions could seem too limiting, we claim that useful information can actually be gathered by analysing appropriate sets of samples, as we describe in Section 4.3.2.3.

4.3.2 Experimental Analysis

In this section, we leverage the attributions provided by the considered techniques to empirically find out the main characteristics of the ransomware samples. After illustrating the experimental setting (Section 4.3.2.1), we perform preparatory tests

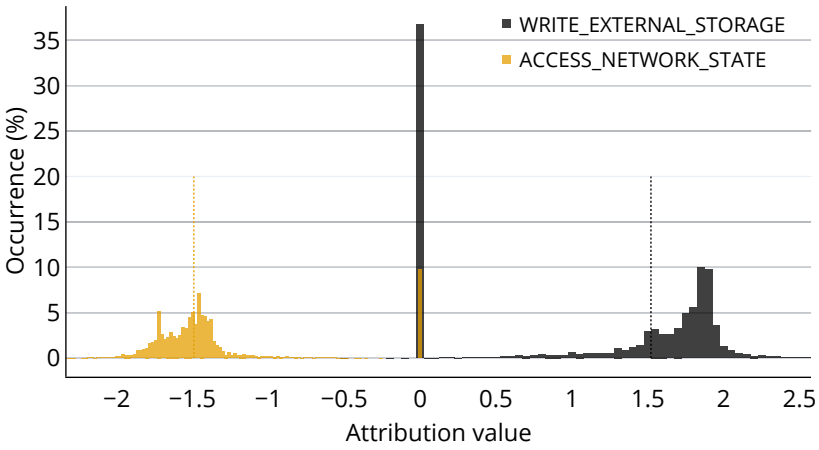


FIGURE 4.9: Attribution distribution of two features, calculated for an MLP classifier through Integrated Gradients. Positive values associate the feature to the ransomware class, negative values to the trusted one. The dotted lines represent the median value.

(Section 4.3.2.2), which enable us to eventually analyse the generated explanations through different criteria (Section 4.3.2.3).

4.3.2.1 Setting

We operate in a two-class setting, where the learning algorithms are trained to classify the positive class of *ransomware* samples against the negative one of *trusted* samples. In the following, I briefly summarise the parameters and the implementation used for the experiments.

Dataset and Features We use the same dataset as [18] (Section 4.2), with 18396 trusted and 1945 ransomware samples, which span from 2011 to 2018 according to their `dex` last modification date. As explained in Section 3.2.1, recent Android versions limit the impact of older ransomware samples; however, since these system updates are installed in a minority of devices, these attacks are still relevant and suitable to assess the validity of our approach. The feature vector consists of 731 features. Among them, 196 represent the occurrence of the API *package* calls. We cumulate all the Android platform APIs until level 26 (Android Oreo). Moreover, the set of APIs is strictly limited to the Android *platform* [86] ones. The remaining amount of features consists of checking the request of permissions extracted from the `AndroidManifest.xml`: when one permission is used, we assign to the correspondent feature a 1; 0 otherwise. In this case, we cumulate the list of permissions until level 29 (Android 10). Since each new API level typically adds new packages, the APIs introduced after 2018 cause the correspondent features to be not used. `aapt`[16] is employed to extract the permissions used by each `APK`. Permissions' names are indicated in capital letters.

Classifiers Since we want our approach to be model-agnostic, we consider three different classifiers: a linear support-vector machine (SVM) and a support-vector

[16] `https://developer.android.com/studio/command-line/aapt2`

machine with RBF kernel (SVM-RBF), both implemented with secml [87], a library for performing adversarial attacks and explaining machine learning models. The third classifier is a multi-layer perceptron (MLP), implemented with Keras. [17] Each classifier has been trained and optimised in its parameters with a repeated 5-fold cross-validation, using 50% of the dataset as the test set. Their detection performance are shown in Section 4.3.2.3.
Notably, these classifiers do not reach the state-of-the-art performance of a Random Forest classifier, which indeed had been used in Section 4.2. The reason for not using it is that this ensemble algorithm presents a non-differentiable decision function; therefore, it is not possible to use gradient-based techniques on top of it. Although it could be possible to train a surrogate classifier to simulate the Random Forest behaviour, it is not necessary to reach state-of-the-art performance in the context of the experiments that follow.

Attribution Computation The attributions are calculated on the best classifier of the first iteration from the cross-validation. To produce the explanations for SVM and SVM-RBF we use secml; for MLP, we use DeepExplain [88]. Since its implementation of Gradient does not return a signed attribution, we switch to iNNvestigate [89] in that case. We compute the attributions with respect to the ransomware class. Consequently, *a positive attribution value always identifies a feature that the classifier associates with ransomware.* As regards Integrated Gradients, since its computation includes an integral, this is approximated through a sum of n parts. We use $n = 130$. As a baseline, a zero-vector is used.

Correlation and Similarity Metrics The experiments of Section 4.3.2.2 make use of three correlation metrics: *Pearson*, *cosine similarity*, *intersection size*. In particular, given two attribution vectors $\boldsymbol{r^1}$ and $\boldsymbol{r^2}$ with d components (which corresponds to the number of features), we consider the median value of each component over a certain set of N samples. Therefore, we obtain two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, where $x_i = median(\boldsymbol{r^1}_i)_N$ and $y_i = median(\boldsymbol{r^2}_i)_N$, for $i = 1, 2, ..., d$. As per the first two correlation metrics, this leads to the following formulation:

$$\text{Pearson}(\boldsymbol{x}, \boldsymbol{y}) := \frac{(\sum_{i=1}^{d} x_i y_i) - d\bar{x}\bar{y}}{\sqrt{(\sum_{i=1}^{d} {x_i}^2) - d\bar{x}^2}\sqrt{(\sum_{i=1}^{d} {y_i}^2) - d\bar{y}^2}} \tag{4.2}$$

$$\text{Cosine Similarity}(\boldsymbol{x}, \boldsymbol{y}) := \frac{\sum_{i=1}^{d} x_i y_i}{\|\boldsymbol{x}\|\|\boldsymbol{y}\|} \tag{4.3}$$

In both cases, the output value lies in the range $[-1, 1]$. Intersection size follows the formulation used by Warnecke et al. [83], which is the following:

$$\text{IS}(\boldsymbol{x}, \boldsymbol{y}) := \frac{|T_x \cap T_y|}{k} \tag{4.4}$$

where T_x and T_y represent the sets of k features with the highest relevance from the attribution vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, respectively. The intersection size lies in the $[0, 1]$

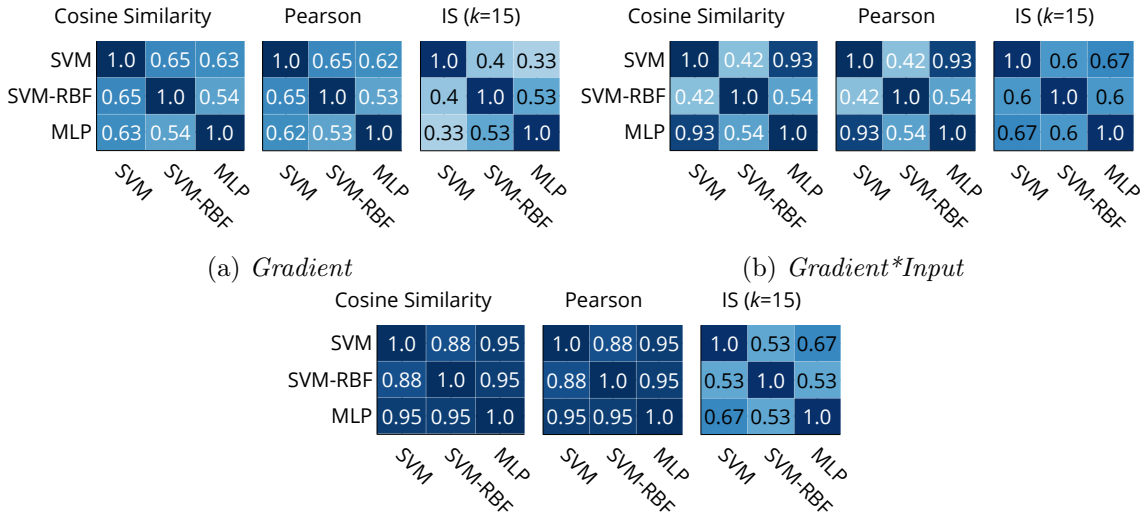
[17] `https://keras.io/`

range, where IS = 0 indicates no overlap and IS = 1 same top-k features. In our case, we choose $k = 15$. This number derives from our manual examination of the attributions since we have observed it represents the typical number of relevant features (i.e. whose values are not zero or close to) for a sample in our setting. Differently from the other two metrics, this one does not consider the sign of the attributions. Therefore, it is useful to catch the importance of a feature regardless of the assignment to a specific class.

4.3.2.2 Preliminary Evaluation

We start our investigation through a set of tests where we compare and correlate median explanation vectors according to different criteria. To make the explanation vectors under test comparable, we scale the attribution values in the range $[-1, 1]$.

Model Influence In this first test, we correlate the attributions of the classifiers with each other, given a fixed explanation method. The results are shown in Figure 4.10. It is possible to notice how the Integrated Gradients case shows particularly-high correlation values, which indicates that its explanations are quite similar to each other across the three considered classifiers. Conversely, this is much less valid for Gradient, where each classifier provides a more dissimilar distribution of the attributions. This first result suggests that, in our setting, the impact of the specific learning algorithm to the explanations is quite modest, especially for the case of Integrated Gradients. In other words, each of the three considered classifiers could be *interchangeably* selected as the reference model for the analysis of the explanations.



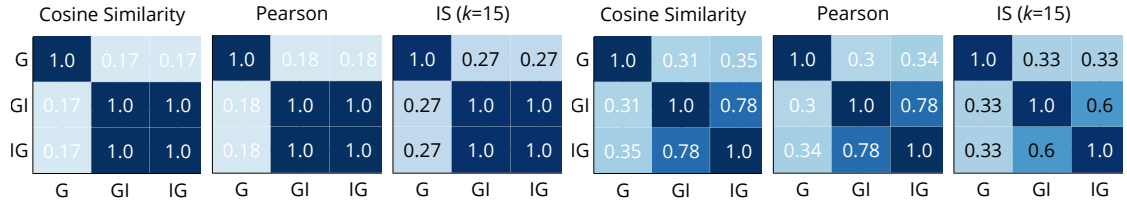
(a) *Gradient* (b) *Gradient*Input*

(c) *Integrated Gradients*

FIGURE 4.10: Model influence: correlation between attributions of each classifier.

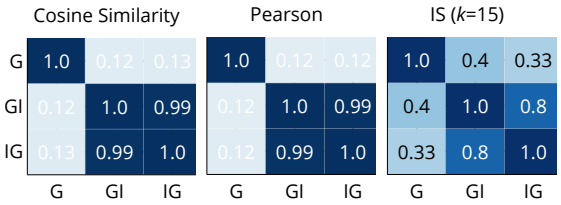
Explanation Method Influence Complementary to the previous experiment, we inspect to what extent the three explanation methods considered are similar. Therefore, for every classifier, we correlate the median attribution vectors of each technique. The results are shown in Figure 4.11. This test highlights the fact that

Gradient*Input and Integrated Gradients are very similar — and also equivalent with linear classifiers. On the contrary, as the previous tests have suggested as well, Gradient produces quite dissimilar explanations.



(a) SVM classifier

(b) SVM-RBF classifier



(c) MLP classifier

FIGURE 4.11: Explanation method influence: correlation between attributions of each technique. G=Gradient, GI=Gradient*Input, IG=Integrated Gradients.

Data Influence Finally, we evaluate the possible impact of the data on the explanations. To do so, we consider randomized labels for the samples of the training set, and we train new classifiers with these labels, forcing a 50% accuracy on the test set. Then we correlate the attribution vectors of these classifiers with those of the original ones. The results, grouped by explainability technique, are visible in Table 4.4. In this case, all the methods present similar results. Notably, only Gradient has nearly-to-zero correlation values, but the Pearson metric exhibits high *p-values*, making the results less reliable. Therefore, we can affirm that all the techniques, included Gradient*Input and Integrated Gradients — besides the presence of the input in their computation — reflect what the model learns, without particularly being bounded by the samples.

TABLE 4.4: Data influence. The Pearson column also includes the p-value in brackets. CS=Cosine Similarity.

Classifier	Gradient			Gradient*Input			Integrated Gradients		
	CS	Pearson	IS ($k = 15$)	CS	Pearson	IS ($k = 15$)	CS	Pearson	IS ($k = 15$)
SVM	0.01	0.02 (0.66)	0.00	0.29	0.29 (0.00)	0.27	0.29	0.29 (0.00)	0.27
SVM-RBF	0.00	-0.01 (0.81)	0.20	0.05	0.04 (0.22)	0.27	-0.22	-0.23 (0.00)	0.07
MLP	0.08	0.06 (0.09)	0.13	0.11	0.10 (0.00)	0.40	0.09	0.09 (0.01)	0.40

Overall, although all the techniques appear to be not that diverse, we have also observed by manual inspection that Gradient tends to produce less *sparse* explanations; i.e. a larger amount of features presents non-null relevance values. Ultimately, we claim the most suitable technique is, for our problem, one between Gradient*Input and Integrated Gradients.

4.3.2.3 Explanation Analysis

In this section, I report the analysis of the explanations. Given the results from the previous section, we examine the explanations provided by one single technique and classifier, assuming that the selected combination is representative of the detection of Android ransomware. Ultimately, *we choose the Integrated Gradients technique and the MLP classifier as such a reference.*

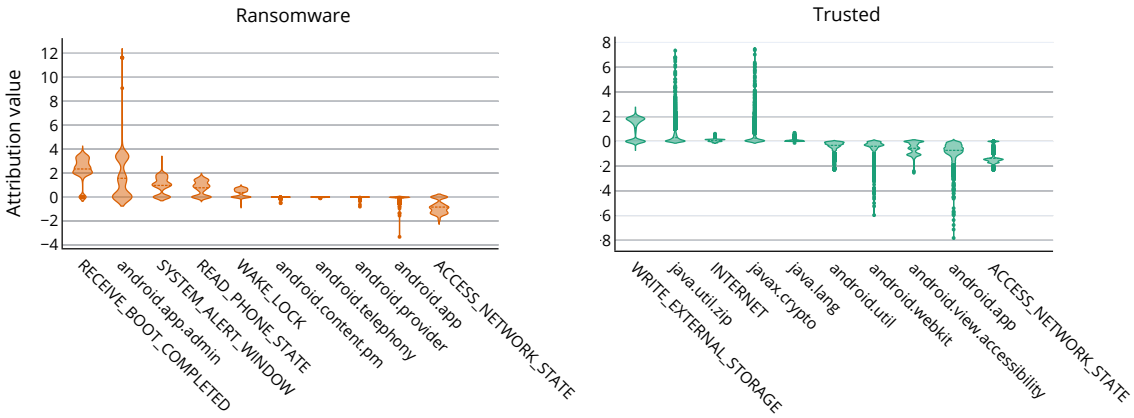
At this point, we aim at understanding how — on average — the features can make a distinction between the trusted and the ransomware class. Moreover, we specifically characterise the behaviour of the ransomware samples and their families by associating the most relevant features to the corresponding action in the app. To do so, we group the samples according to different criteria. Differently from Section 4.3.2.2, the attributions are not normalised; consequently, there is no lower or upper bound to consider as a reference for the magnitude of each attribution. This choice also preserves the possibility to sort and compare them across the different features. If not stated differently, we consider the attributions calculated on all the dataset samples.

Evaluation by Class We consider the average explanations separately for the trusted and ransomware class. In particular, we sort the attribution values of each group according to the median value of each feature, and we inspect the top features with positive values and the top ones with negative values. In this way, we can inspect how the features gravitate towards one of the two classes and revealing the kind of behaviour they can be associated with. Table 4.5 exemplifies the interpretation of the four possibile expected behaviours.

TABLE 4.5: Behaviour associated with each feature depending on the sign of the attribution values and the samples used to calculate the explanations.

	Trusted sample	**Ransomware sample**
Positive attribution	Non-trusted	Ransomware
Negative attribution	Trusted	Non-ransomware

Figure 4.12 shows the distribution of the attribution values for the top-5 positive and top-5 negative relevant features of the ransomware (a) and trusted (b) classes. As a first observation, we can notice that the highest median values are associated with the features that go into the direction of the ransomware class, while trusted samples' attributions exhibit lower sparsity and much higher variance. This fact suggests that trusted samples need a higher number of features to have them described, being the set of apps much broader and diversified. Going into more detail, let us consider Figure 4.12a. The top-5 positive features can be reasonably associated with the behaviour of a generic malware. For example, `RECEIVE_BOOT_COMPLETED` enables an app to start after a device reboot, while `WAKE_LOCK` avoids it being killed by the operating system. Moreover, we can see the presence of a *ransomware-specific* feature — `SYSTEM_ALERT_WINDOW` — that is a permission that allows the attacker to display an overlay window (see Section 3.2.1), and that is often tied with `android.app.admin`. The top-5 negative features, such as `ACCESS_NETWORK_STATE`, should be interpreted as typical of *non-ransomware* apps. Concerning the trusted samples, Figure 4.12b shows as the most



(a) (b)

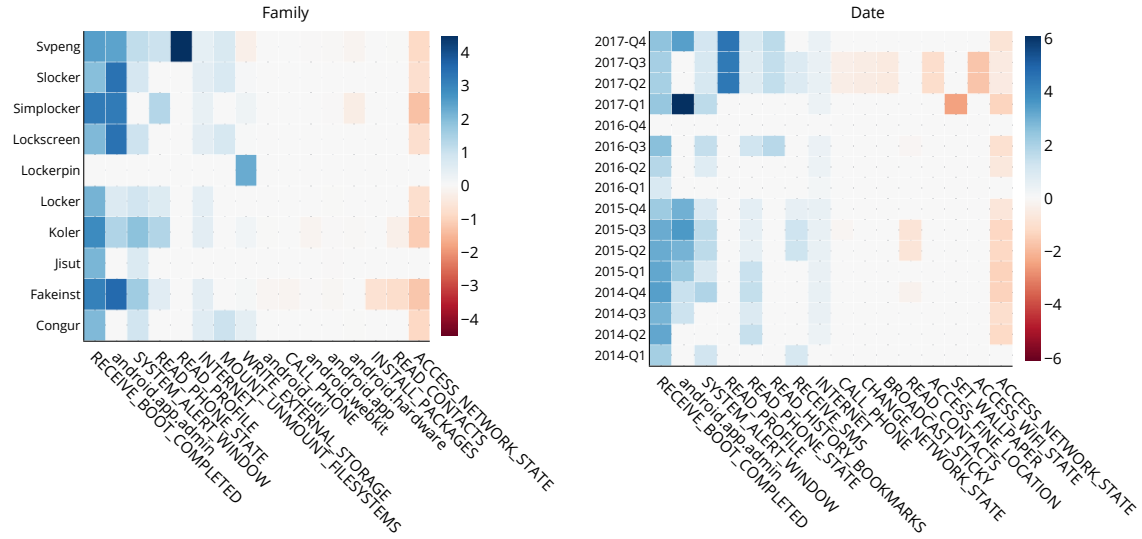
FIGURE 4.12: Top-5 positive and top-5 negative feature attribution distribution for the ransomware (a) and trusted (b) samples of the dataset.

prominent feature `WRITE_EXTERNAL_STORAGE`, a permission that can be intuitively associated with crypto-ransomware apps. Other positive values (*non-trusted* features) that emerge are `javax.crypto` and `java.security`, which are characteristic of crypto-ransomware apps as well. Among the *trusted-specific* features, an illustrative example comes from `android.app`, which provides a set of layout components that a ransomware developer does not use broadly.

Evaluation by Ransomware Family Focusing on ransomware applications, we now inspect to what degree their different families exhibit shared and peculiar traits. AVClass [77] is used to extract the plausible families from the VirusTotal reports of each sample. Figure 4.13a shows the median values for the attributions of the main ransomware families of the dataset. We limit the analysis to the families with at least 30 samples, obtaining ten of them. As can be noticed by looking along the vertical axis of this plot, some features exhibit a stable relevance level across most of the families. In other words, features like `SYSTEM_ALERT_WINDOW` can be considered to be a common characteristic of all ransomware apps. Other ones seem to be peculiar of specific families.

Looking at the properties of each family (rows of the plot), possible peculiarities can be noticed, such as for `Svpeng` and `Lockerpin`. In the first case, the family presents a strongly positive relevance of `READ_PROFILE`, while the `Lockerpin` family's attributions seem to exhibit zero relevance values instead, except for the `WRITE_EXTERNAL_STORAGE` permission. To investigate the reason for that, we have looked for representative samples by picking the ones that were closest to the median attributions, using the `cosine similarity` function. For the `Svpeng` family, we have obtained a locker ransomware sample[18] that, after hiding as a porn app, pretends to be the FBI enforcement and shows the victims messages that include the threat to send a supposed criminal record to their contacts, which are explicitly shown in the message. These contacts are gathered through the `READ_PROFILE` permission, which explains its relevance. In the second case[19], the anomalous explanations are because the app contains, within its `assets`, another `APK` file that gets

[18] MD5: `8a7fea6a5279e8f64a56aa192d2e7cf0`
[19] MD5: `1fe4bc42222ec1119559f302ed8febfc`



(a) (b)

FIGURE 4.13: Top positive and negative attributions' median values for two grouping criteria: family (a) and date (b).

installed after the user opens the original app and grants the requested privileges. Therefore, the original app is merely a container.

Evaluation by Ransomware Date The analysis of the evolution of malware attacks is particularly relevant for machine learning-based detection systems. As a matter of fact, since they do not employ signatures to identify malicious behaviour, they are often able to detect new variants of previously observed families. At the same time, they should be carefully evaluated over time to avoid experimental bias [90]. Therefore, it could be useful to understand what features do and do not possess a certain level of relevance, regardless of the ransomware evolution. To do so, we extract the last modification date from the `dex` file of each `APK`. We discard the samples with non-plausible dates (e.g. the ones with a *Unix epoch* date), and we group the remaining ones in windows of three months. The result is shown in Figure 4.13b. As can be noticed, some features maintain pretty much the same relevance values over time, which makes them *resilient* to new ransomware variants. In other words, some features describe essential components for ransomware samples. It should be noticed that also the types of features we employ — API calls and permission requests — change over time according to the Android development; therefore, both the attacker and the detector designer have to adapt to this progression. Figure 4.13b also shows that the relevance of other features could depend on the spread of a particular family. For example, `READ_PROFILE` starts being relevant in 2017, when the previously-described `Svpeng` appeared.

Reduced Feature Set Evaluation We finally inspect if the explanations analysed in the previous experiments, after helping with characterising ransomware apps, can be used to change the group of features of the system under design. In other words, we aspire to build a feature set that, although it might cause accuracy loss, minimizes the learning of spurious patterns by the classifier. To do so, we construct a reduced feature set composed of the top-20 relevant attributions (10

positive and 10 negative) for both classes. Notably, we still use the attributions from Integrated Gradients for the MLP classifier, but we only consider training samples. We attain a set of 33 unique features. Figure 4.14 shows a ROC curve with a comparison between the original three classifiers, trained with the full feature set, and the correspondent ones trained with this reduced set of features. Besides the minimal number of features, the results at the threshold of FPR=1% are not so far from those with the full feature set (where 353 features out of all 731 ones are used with our dataset). Only the linear SVM sees quite a significant drop in the detection rate.

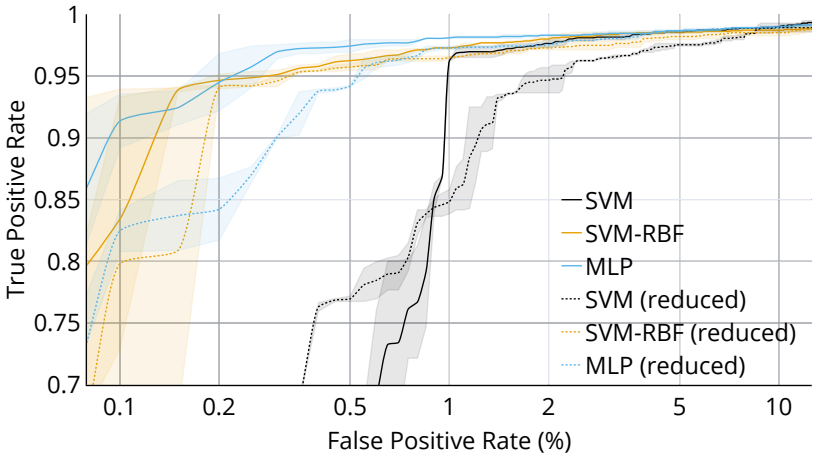


FIGURE 4.14: Roc curve comparison. The classifiers with solid lines have been trained with the full feature set, the ones with dotted lines with a reduced feature set made of 33 features.

4.3.3 Contributions, Limitations, and Future Work

Through this work, we have presented an initial proposal for a method that eases designers in finding the features that mostly characterise Android ransomware. First, we have developed a set of empirical tests that allows selecting, among several combinations of explanation methods and classifiers, a unique setting that best represents the domain under analysis. In this way, human experts are able to more easily investigate the output produced by such techniques. Second, by looking at the *average* importance of the features within domain-specific sets of samples, we have shown the Android ransomware's main traits that effectively distinguish malicious samples from legitimate ones, and the ransomware's evolution over time in terms of application components. This novel work has corroborated the previous evidence that API calls effectively catch the peculiar actions operated by ransomware apps, with a restricted set of APIs revealing *ransomware-specific* behaviour, while several others identifying *non-ransomware* apps. Moreover, permissions have turned out to be quite impactful as well.

Being this a preliminary proposal and given the changes in the setting with respect to the one of Section 4.2, a few limitations should be highlighted. First, we claim that the two-class system we have considered, where ransomware apps are evaluated against legitimate ones, does not fully capture behaviour *in the middle*. In other words, we are not able to identify the characteristics that ransomware attacks

do and do not share with generic malware. Moreover, we have used requested permissions as additional features, while at the same time not considering System API *class* or *method* calls. Therefore, we believe that further new insights can be revealed by considering other setting variants. For example, we could check if classifiers using finer-grained API calls provide coherent explanations with the ones using packages.
Another limitation is related to the usage of attribution techniques, i.e. the fact that it is not possible to understand how the features — especially of different types as in this specific setting — interact with each other. This is one of the open issues in the field of explainable machine learning [40, 41], and we think discovering high-level *concepts* might greatly improve the understanding of the applications under analysis.
We expect that the proposed approach could be refined by making it more systematic and, above all, further inspecting the practical difference between each gradient-based technique. In this way, it could also be possible to *combine* the complementary peculiarities of each one, instead of selecting a single technique. Moreover, although our work has shown that gradient-based explanation methods are quite effective against the considered feature space, it could be beneficial to broaden the focus to other attribution techniques. Lastly, being Android ransomware one of the possible case studies, we point out the possibility to generalise our method to other malware detection problems.

Chapter 5

Explainable and Adversarial Machine Learning

The previous chapter has shown that learning-based techniques based on static analysis are especially effective at detecting Android malware, which constitutes one of the major threats in mobile security. In particular, the described approach, along with the ones in the literature, show great accuracy even when traditional code concealing techniques (such as static obfuscation) are employed [15–18].
Despite the successful results reported by such approaches, the problem of detecting malware created to fool learning-based systems is still far from being solved. The robustness of machine-learning models is challenged by the creation of *adversarial examples* [20, 27–29] (see Section 2.1). In particular, recent work concerning Android malware has demonstrated that specific changes to the contents of malicious Android applications might suffice to change their classification (e.g. from malicious to benign) [17, 91]. The main characteristic of these attacks is their *sparsity*, meaning that they enforce only a few changes to the whole feature set to be effective. Such changes may be represented by, for example, the injection of unused permissions or parts of unreachable or unused executable code. For example, adding a component that is loaded when the application starts (through a keyword called `LAUNCHER`) can significantly influence the classifier's decision [81].

On the basis of such issues, this chapter presents the effort to address them through two different points of view, both sharing the usage of gradient-based explanation techniques. The first one (Section 5.1, based on the work by Melis et al. [92]) aims to establish a way to assess through explanations the expected vulnerability of classifiers to evasion attacks. The second work (Section 5.2, based on the article by Cara et al. [93]) has the main goal of studying the practical feasibility of creating Android adversarial samples through the injection of System API calls.

5.1 Do Explanations Tell Anything About Adversarial Robustness?

One of the many reasons why adversarial attacks against Android malware detectors are so effective is that classifiers typically assign significant relevance to a limited amount of features (this phenomenon has also been demonstrated in other applications such as email spam filtering). As a possible countermeasure, research has

shown that classifiers that avoid overemphasising specific features, weighting them more evenly, can be more robust against such attacks [17, 94, 95]. Simple metrics characterising this behaviour were proposed to identify and select more robust algorithms, especially in the context of linear classifiers, where feature weights can be used as a direct measure of a feature's relevance to each decision [17, 96, 97]. In parallel, the ability to understand the classifiers' behaviour by looking at the input gradient, i.e. the feature weights in the case of linear classifiers, was also explored by multiple works in the field of explainable machine learning [33, 46–48]. In particular, it became of interest to figure out if the information provided by these gradient-based methods can also be employed to understand (and improve) the robustness of learning-based systems against attacks [98].

In the following, I present our effort to investigate the possible correlations between gradient-based explanations and the classifiers robustness to adversarial evasion attacks on an Android malware detection case study. We assess our findings on Drebin, a popular learning-based detector for Android (Section 5.1.1). I first provide a description of the adversarial vulnerabilities of learning-based systems for Android malware detection (Section 5.1.2). Then, motivated by the intuition that the classifiers whose attributions are more evenly distributed should also be the more robust, as they rely on a broader set of features for the decision, we propose and empirically validate a few synthetic metrics that allow correlating between the *evenness* of gradient-based explanations and the *adversarial robustness* — a measure we propose to represent the classifier robustness to adversarial attacks along with an increasing attack power in a compact way (Section 5.1.3). Our investigation (Section 5.1.4) unveils that, under some circumstances, there is a clear relationship between the distribution of gradient-based explanations and the adversarial robustness of Android malware detectors. I finally make concluding remarks on how our findings can pave the way towards the development of more efficient mechanisms both to evaluate adversarial robustness and to defend against adversarial Android malware examples (Section 5.1.5).

5.1.1 Background on Drebin

The majority of the approaches for Android malware detection employ static and dynamic analyses that extract information such as permissions, communications through Inter-Component Communication (ICC), system- and user-implemented API calls, and so forth [15, 16, 18, 99, 100].

Drebin is among the most popular and used static detection approaches. It performs the detection of Android malware through static analysis of Android applications. In a first phase (training), it employs a set of benign and malicious apps provided by the user to determine the features that will be used for detection (meaning that the feature set will be strictly dependent on the training data). Such features are then embedded into a *sparse*, high-dimensional vector space. Then, after the training of a linear machine-learning model, the system is able to perform the classification of previously-unseen apps. An overview of the system architecture is given in Figure 5.1, and discussed more in detail below.

Feature extraction. First, Drebin statically analyses a set of n training Android applications to construct a suitable feature space. All features extracted by

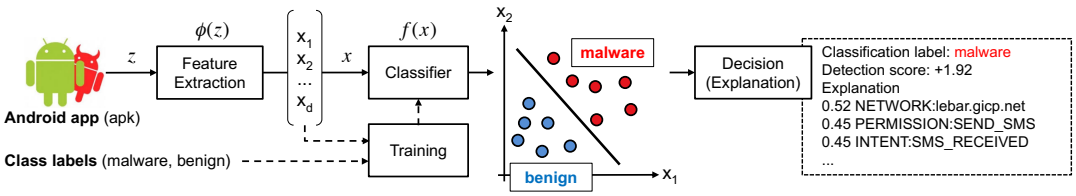


FIGURE 5.1: A schematic representation ([17]) of Drebin. First, applications are represented as binary vectors in a d-dimensional feature space. A linear classifier is then trained on an available set of malware and benign applications, assigning a weight to each feature. During classification, unseen applications are scored by the classifier by summing up the weights of the present features: if $f(\boldsymbol{x}) \geq 0$, they are classified as malware. Drebin also explains each decision by reporting the most suspicious (or benign) features present in the app, along with the weight assigned to them by the linear classifier [15].

	`manifest`		`dexcode`
S_1	Hardware components	S_5	Restricted API calls
S_2	Requested permissions	S_6	Used permission
S_3	Application components	S_7	Suspicious API calls
S_4	Filtered intents	S_8	Network addresses

TABLE 5.1: Overview of feature sets.

Drebin are presented as *strings* and organised in 8 different feature sets, as listed in Table 5.1.

Android applications are then mapped onto the feature space as follows. Let us assume that an app is represented as an object $\boldsymbol{z} \in \mathcal{Z}$, being $\mathcal{Z}$ the abstract space of all `APK` files. We denote with $\Phi : \mathcal{Z} \mapsto \mathcal{X}$ a function that maps an `APK` file $\boldsymbol{z}$ to a d-dimensional feature vector $\boldsymbol{x} = (x^1, \ldots, x^{\mathsf{d}})^\top \in \mathcal{X} = \{0,1\}^{\mathsf{d}}$, where each feature is set to 1 (0) if the corresponding *string* is present (absent) in the `APK` file $\boldsymbol{z}$. An application encoded in feature space may then look like the following:

$$\boldsymbol{x} = \Phi(\boldsymbol{z}) \mapsto \begin{pmatrix} \cdots \\ 0 \\ 1 \\ \cdots \\ 1 \\ 0 \\ \cdots \end{pmatrix} \begin{matrix} \cdots \\ \texttt{permission::SEND\_SMS} \\ \texttt{permission::READ\_SMS} \\ \cdots \\ \texttt{api\_call::getDeviceId} \\ \texttt{api\_call::getSubscriberId} \\ \cdots \end{matrix} \begin{matrix} \\ \left.\begin{matrix} \\ \\ \end{matrix}\right\} S_2 \\ \\ \left.\begin{matrix} \\ \\ \end{matrix}\right\} S_5 \\ \end{matrix}$$

Learning and Classification. Drebin uses a linear Support Vector Machine (SVM) to perform detection. It can be expressed in terms of a linear function $f : \mathcal{X} \mapsto \mathbb{R}$, i.e. $f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x} + b$, where $\boldsymbol{w} \in \mathbb{R}^{\mathsf{d}}$ denotes the vector of *feature weights*, and $b \in \mathbb{R}$ is the so-called *bias*. These parameters, optimised during training, identify a hyperplane that separates the two classes in the feature space. During classification, unseen apps are then classified as malware if $f(\boldsymbol{x}) \geq 0$, and as benign otherwise. In this work, we will also consider other linear and nonlinear algorithms to learn the classification function $f(\boldsymbol{x})$.

Explanation. Drebin explains its decisions by reporting, for any given application, the most influential features, i.e. the ones that are present in the given application and are assigned the highest absolute weights by the classifier. The feature relevance values reported by Drebin correspond exactly to its feature weights, being Drebin a linear classifier. For instance, in Figure 5.1 it is possible to see that Drebin correctly identifies the sample as malware since it connects to a suspicious URL and uses SMS as a side-channel for communication. In this work, we use different explanation approaches to measure feature relevance and evaluate whether and to which extent the distribution of relevance values reveals any interesting insight on adversarial robustness.

5.1.2 Adversarial Android Malware

Machine learning algorithms are known to be vulnerable to adversarial examples. The ones used for Android malware detection do not constitute an exception. The vulnerability of those systems has been demonstrated in [17, 32, 97], and a defence mechanism has been proposed in [17]. In this section, I first explain how an attacker can construct Android malware able to fool a classifier (Drebin), being recognised as benign. Then, considering the Sec-SVM algorithm [17] as a case-study, I explain how machine learning systems can be strengthened against this attack.

5.1.2.1 Attacking Android Malware Detection

The goal of creating adversarial Android malware that evades detection can be formulated as an optimisation problem, as detailed below. This optimisation problem is constrained to ensure that the solution provides a functional and feasible malware sample, i.e. that the feature changes suggested by the attack algorithm are feasible and can be implemented as practical manipulations to the actual `APK` input file.

Problem Formulation As introduced in Section 5.1.1, Drebin is a binary classifier trained on boolean features. To have a malware sample $\boldsymbol{z}$ misclassified as benign, the attacker should modify its feature vector $\boldsymbol{x}$ in order to decrease the classifier score $f(\boldsymbol{x})$. The number of features considered by Drebin is quite large (more than one million). However, the attacker can reasonably change only few of them (*sparse attack*) to preserve the malicious functionality of the application. The attacker has then an ℓ_1-norm constraint on the number of features that can be modified. The feature vector of the adversarial application can be computed by solving the following optimisation problem:

$$\begin{aligned} \arg\min_{\boldsymbol{x}'} \quad & f(\boldsymbol{x}') && (5.1)\\ \text{s.t.} \quad & \|\boldsymbol{x} - \boldsymbol{x}'\|_1 \leq \varepsilon && (5.2)\\ & \boldsymbol{x}_{\text{lb}} \preceq \boldsymbol{x}' \preceq \boldsymbol{x}_{\text{ub}} && (5.3)\\ & \boldsymbol{x}' \in \{0,1\} \quad , && (5.4) \end{aligned}$$

where Equation (5.2) is the ℓ_1 distance constraint between the original $\boldsymbol{x}$ and the modified (adversarial) $\boldsymbol{x}'$ sample. Equation (5.3) is a box constraint that enforces the adversarial malware's feature values to stay within some lower and upper bounds,

Algorithm 1 PGD-based attack on Android malware.

Input: $\boldsymbol{x}$, the input malware; ε, the number of features which can be modified; η, the step size; Π, a projection operator on the constraints (5.2) and (5.3); $t > 0$, a small number to ensure convergence.

Output: $\boldsymbol{x}'$, the adversarial (perturbed) malware.

$\boldsymbol{x}' \leftarrow \boldsymbol{x}$
repeat
  $\boldsymbol{x}^\star \leftarrow \boldsymbol{x}'$
  $\boldsymbol{x}' \leftarrow \Pi(\boldsymbol{x}^\star - \eta \cdot \nabla f(\boldsymbol{x}^\star))$
until $|f(\boldsymbol{x}') - f(\boldsymbol{x}^\star)| \leq t$
return: $\boldsymbol{x}'$

while Equation (5.4) enforces the attack to find a Boolean solution. The aforementioned problem can be solved with gradient-based optimisation techniques, e.g. Projected Gradient Descent (PGD), as described in Algorithm 1 [27, 97, 101]. At each step, this algorithm projects the feature values of the adversarial sample onto the constraints (Equations (5.2)–(5.3)), including binarisation in $\{0, 1\}$.

Feature Addition To create malware able to fool the classifier, an attacker may, in theory, both adding and removing features from the original applications. However, in practice, removing features is a non-trivial operation that can easily compromise the malicious functionalities of the application. Feature addition is a safer operation, especially when the injected features belong to the `manifest`; for example, adding permissions does not influence any existing application functionality. When the features depend on the `dexcode`, it is possible to add them safely introducing information that is not actively executed, e.g. by adding code after `return` instructions (*dead code*) or methods that are never called by any `invoke` type instructions (i.e. the ones that indicate a method call). Therefore, in this work, we only consider feature addition. To find a solution that does not require removing features from the original application, the attacker can simply define $\boldsymbol{x}^{lb} = \boldsymbol{x}$ in Equation (5.3). However, it is worth mentioning that this injection could be easily made ineffective, simply removing all the features extracted from code lines that are never executed. In this way, the attacker is forced to change the executed code, which is more difficult, as it requires considering the following additional and stricter constraints. Firstly, the attacker should avoid breaking the application functionalities. Secondly, they should avoid introducing possible artefacts or undesired functionalities, which may influence the semantics of the original program. Injecting a large number of features may be, therefore, difficult and not always feasible. This aspect will be discussed further later in this chapter (Section 5.2).

5.1.2.2 SecSVM: Defending against Adversarial Android Malware

In [17], the authors have shown that the sparse evasion attack described above is able to fool Drebin, requiring the injection of a negligible number of features, and they have proposed a robust counterpart of that classifier. The underlying idea behind their countermeasure is to enforce the classifier to learn more evenly distributed feature weights since this will require the attacker to manipulate more features to evade the classifier. To this end, they have added a box constraint on the weights

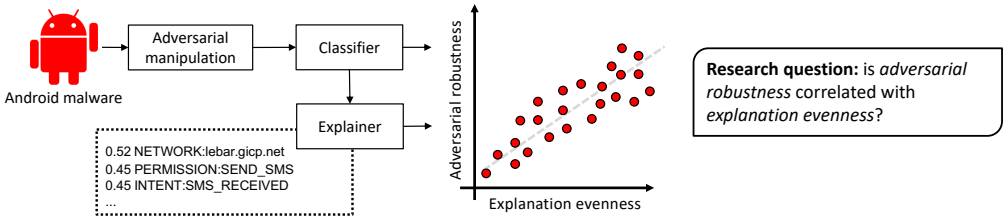


FIGURE 5.2: Schematic representation of the analysis employed to verify the correlation between explanation evenness and adversarial robustness. First, for each malware in the test set, we create its adversarial counterpart. Then, for each of those adversarial applications, we evaluate: (1) a measure of the classifier robustness against it (*adversarial robustness*) (2) the evenness of the application attributions (*explanation evenness*). Finally, we asses the correlation between them.

$\boldsymbol{w}$ of a linear SVM, obtaining the following learning algorithm (Sec-SVM):

$$\begin{aligned} \min_{\boldsymbol{w},b} \quad & \tfrac{1}{2}\boldsymbol{w}^\top \boldsymbol{w} + C\sum_{i=1}^{\mathsf{n}} \max\left(0, 1 - y_i f(\boldsymbol{x}_i)\right) \\ \text{s.t.} \quad & w_k^{\text{lb}} \leq w_k \leq w_k^{\text{ub}}\,,\ k = 1, \ldots, \mathsf{d} \quad , \end{aligned} \tag{5.5}$$

where the lower and upper bounds on $\boldsymbol{w}$ are defined by the vectors $\boldsymbol{w}^{\text{lb}} = (w_1^{\text{lb}}, \ldots, w_{\mathsf{d}}^{\text{lb}})$ and $\boldsymbol{w}^{\text{ub}} = (w_1^{\text{ub}}, \ldots, w_{\mathsf{d}}^{\text{ub}})$, which are application dependent. Section 5.1.2.2 can be easily optimised using a constrained variant of the Stochastic Gradient Descent (SGD) technique, as described in [17].

5.1.3 Do Gradient-based Explanations Help to Understand Adversarial Robustness?

In this work, we investigate whether gradient-based attribution methods used to explain classifiers' decisions provide useful information about the robustness of Android malware detectors against sparse attacks. Our intuition is that the classifiers whose attributions are usually evenly-distributed rely upon a broad set of features instead of overemphasising only a few of them. Therefore, they are more robust against sparse attacks, where the attacker can change only few features, having a negligible impact on the classifier decision function. To verify our intuition, we present an empirical analysis whose procedure is illustrated in Figure 5.2 and described below. Firstly, we perform a security evaluation on the tested classifier, obtaining a compact measure we call *Adversarial Robustness* (see Section 5.1.3.1), representing its robustness to the adversarial attacks along with an increasing number of added features ϵ. Then, we compute the attributions for each benign and manipulated malware sample $\boldsymbol{x}$ using gradient-based explanation techniques, obtaining the relevance vectors $\boldsymbol{r}$. For each of those, we propose to look for a compact metric that encapsulates the degree of *Evenness* of the attributions (see Section 5.1.3.2). Finally, comparing this value with the adversarial robustness, we assess the connection between attributions' evenness and the robustness to adversarial evasion attacks.

5.1.3.1 Adversarial Robustness

We define the robustness to the evasion samples crafted injecting a fixed number of features ϵ as:

$$R(\mathcal{D}_\varepsilon, f) = \frac{1}{n}\sum_{i=1}^{n} e^{-\ell_i} \quad , \tag{5.6}$$

where $\ell_i = \ell(y_i, f(\boldsymbol{x}_i))$ is the adversarial loss attained by the classifier f on the data points in $\mathcal{D}_\varepsilon = \{\boldsymbol{x}_i, y_i\}_{i=1}^n$, containing the ε-sized adversarial samples optimised with Algorithm 1.

We finally define the adversarial robustness $\mathcal{R}$ of a classifier f as the average of $R(\mathcal{D}_\varepsilon, f)$ on different ε:

$$\mathcal{R} = \mathbb{E}_\varepsilon\{R(\mathcal{D}_\varepsilon, f)\} \quad . \tag{5.7}$$

5.1.3.2 Explanation Evenness Metrics

To compute the evenness of the attributions, we consider the two metrics, described below. The first is the one proposed in [94, 95]. To compute the evenness metric, they first define a function $F(\boldsymbol{r}, k)$ that, given a relevance vector $\boldsymbol{r}$, computes the ratio of the sum of the k highest relevance values to the sum of all absolute relevance values, for $k = 1, 2, \ldots, m$:

$$F(\boldsymbol{r}, k) = \frac{\sum_{i=1}^{k} |r_{(i)}|}{\sum_{j=1}^{m} |r_{(j)}|} \quad ,$$

where $r_1, r_2, \ldots, r_m$ denote the relevance values, sorted in descending order of their absolute values, i.e. $|r_1| \geq |r_2| \geq \ldots \geq |r_m|$ and m is the number of considered relevance values ($m \leq d$). This function essentially computes the evenness of the distribution of the relevance among the features. The evenest relevance distribution (the one where they are all equal), corresponds to $F(\boldsymbol{r}, k) = k/n$. Whereas the most uneven is attained when only one relevance differs from zero, and in this case, $F(\boldsymbol{r}, k) = 1$ for each k value. To avoid the dependence on k and to obtain a single scalar value, they compute the evenness as:

$$\mathcal{E}_1(\boldsymbol{r}) = \frac{2}{m-1}\left[m - \sum_{k=1}^{m} F(\boldsymbol{r}, k)\right] \quad . \tag{5.8}$$

The range of $\mathcal{E}_1$ is $[0, 1]$, $\mathcal{E}_1 = 0$ and $\mathcal{E}_1 = 1$ indicates respectively to the most uneven and to the most even relevance vector.

The second metric we consider is the one proposed in [96], based on the ratio between the ℓ_1 and ℓ_∞ norm:

$$\mathcal{E}_2(\boldsymbol{r}) = \frac{1}{m} \cdot \frac{\|\boldsymbol{r}\|_1}{\|\boldsymbol{r}\|_\infty} \quad . \tag{5.9}$$

To have a broader perspective of the attributions evenness, we compute the metrics on multiple samples, and we average the results. More formally, we define the *explanation evenness* as:

$$E = \frac{1}{n}\sum_{i=1}^{n} \mathcal{E}(\boldsymbol{r}^i) \quad , \tag{5.10}$$

where $\boldsymbol{r}^i$ with $i = 1, 2, \ldots, n$ is the relevance vector computed on each sample of a test dataset $\mathcal{D} = \{\boldsymbol{x}_i, y_i\}_{i=1}^n$, and $\mathcal{E}$ can be equal either to $\mathcal{E}_1$ or $\mathcal{E}_2$. In the following, we represent the average evenness computed considering the per-sample metric $\mathcal{E}_1$ ($\mathcal{E}_2$) with E_1 (E_2).

5.1.4 Experimental Analysis

In this section, we practically evaluate whether the measures introduced in Section 5.1.3 can be used to estimate the robustness of classifiers against sparse evasion attacks. After detailing our experimental setup (Section 5.1.4.1), I show the classifiers' detection performances, both in normal conditions and under attack (Section 5.1.4.2). In our evaluations, we focus on the feature addition attack setting (see Section 5.1.2), as they are typically the easiest to accomplish for the adversary. We use secml as a framework to implement classification systems, explanation techniques, and attack algorithms [87]. Finally, we assess the relationship of the proposed evenness metrics with adversarial robustness and detection rate (Section 5.1.4.3).

5.1.4.1 Experimental Setup

Dataset We use the Drebin dataset [15], consisting of 121329 benign applications and 5615 malicious samples, labelled with VirusTotal. A sample is labelled as malicious if it is detected by at least five anti-virus scanners, whereas it is labelled as benign otherwise.

Training-validation-test splits We average our results on 5 runs. In each run, we have randomly selected 60,000 apps from the Drebin data to train the learning algorithms, and we have used the remaining apps for testing.

Classifiers We compare the standard Drebin implementation based on a linear Support Vector Machine (SVM) against the *secured* linear SVM from Section 5.1.2.2 (Sec-SVM), an SVM with the RBF kernel (SVM-RBF), a logistic regression (logistic), and a ridge regression (ridge).

Parameter setting Using a 5-fold cross-validation procedure, we have optimised the parameters of each classifier to maximise the detection rate (i.e. the fraction of detected malware) at 1% false-positive rate (i.e. the fraction of legitimate applications misclassified as malware). In particular, we have optimised $C \in \{10^{-2}, 10^{-1}, \ldots, 10^{2}\}$ for both linear and non-linear SVMs and logistic, the kernel parameter $\gamma \in \{10^{-4}, 10^{-3}, \ldots, 10^{2}\}$ for the SVM-RBF, and the parameter $\alpha \in \{10^{-2}, 10^{-1}, \ldots, 10^{2}\}$ for ridge. For Sec-SVM, we have optimised the parameters $-\boldsymbol{w}^{\mathrm{lb}} = \boldsymbol{w}^{\mathrm{ub}} \in \{0.1, 0.25, 0.5\}$ and $C \in \{10^{-2}, 10^{-1}, \ldots, 10^{2}\}$. When similar detection rates ($\pm 1\%$) were obtained for different hyperparameter configurations, we have selected the configuration corresponding to a more regularised classifier, as more regularised classifiers are expected to be more robust under attack [97]. The typical values of the aforementioned hyperparameters found after cross-validation are $C = 0.1$ for SVM, $\alpha = 10$ for ridge, $C = 1$ for logistic, $C = 1$ and $w = 0.25$ for Sec-SVM, $C = 10$ and $\gamma = 0.01$ for SVM-RBF.

Attribution computation We compute the attributions on 1000 malware samples randomly chosen from the Drebin test set. We take $\boldsymbol{x}' = 0$ as the baseline for Integrated Gradients, and we compute the attributions with respect to the malware class. As a result, positive (negative) relevance values in our analysis denote malicious (benign) behaviour. Given the high sparsity ration of the Drebin dataset, we use $m = 1000$ to compute the explanation evenness metrics.

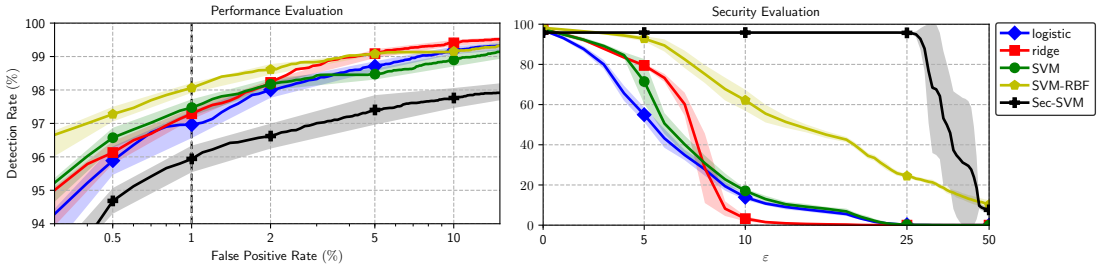


FIGURE 5.3: (left) Mean ROC curves for the tested classifiers on the Drebin data. (right) White-box evasion attacks on Drebin data. Detection Rate at 1% False Positive Rate against an increasing number of added features ε. We can see how the Sec-SVM, although it provides a slightly lower detection rate compared to the other tested classifiers, requires on average more than 25 different new feature additions to the original apps to be fooled by the attacker.

5.1.4.2 Experimental Results

We first perform an evaluation of the performances under normal conditions; the resulting Receiver Operating Characteristic (ROC) curve with the Detection Rate for each classifier, averaged over the 5 repetitions, is reported in the left side of Figure 5.3. We then perform a white-box evasive attack against each classifier, aiming to have 1000 malware samples randomly chosen from the Drebin dataset misclassified as benign. The results are shown on the right side of Figure 5.3, which reports the variation of the detection rate as the number of modified features ε increases. We can notice how the Sec-SVM classifier provides a slightly worse detection rate compared to the other classifiers, but is particularly robust against adversarial evasion attacks.

5.1.4.3 Is adversarial robustness correlated with explanation evenness?

We here investigate the connection between adversarial robustness and evenness of gradient-based explanations. I start with two illustrative examples. Table 5.2 shows the top-10 influential features for two malware samples[1] of the `FakeInstaller` and `Plankton` families, reported for the SVM-RBF and Sec-SVM algorithms, and obtained through the Gradient*Input technique. All the classifiers correctly label the samples as malware.

Looking at the features of the first sample, the `FakeInstaller` malware, we can observe how both the classifiers identify the cellular- and SMS-related features, e.g. the `GetNetworkOperator()` method or the `SEND_SMS` permission, as highly relevant. This is coherent with the actual behaviour of the malware sample since its goal is to send SMS messages to premium-rate numbers. With respect to the relevance values, the first aspect to point out comes from their relative magnitude, expressed as a percentage in Table 5.2. In particular, we can observe that the top-10 relevance values for SVM-RBF vary, regardless of their signs, from 3.49% to 10.35%, while for Sec-SVM the top values lie in the 3.39%–3.51% range. This suggests that SVM-RBF assigned high prominence to few features; conversely, Sec-SVM distributed the relevance values more evenly. It is possible to catch this behaviour more easily through the synthetic evenness measures $\mathcal{E}_1$ (Equation (5.8)) and $\mathcal{E}_2$ (Equation (5.9))

[1] MD5: f8bcbd48f44ce973036fac0bce68a5d5 (`FakeInstaller`) and eb1f454ea622a8d2713918b590241a7e (`Plankton`).

SVM-RBF ($\mathcal{E}_1 = 46.24\%$, $\mathcal{E}_2 = 22.47\%$, $\varepsilon_{\min} = 6$)

Set	Feature Name	r (%)
S2	SEND_SMS	10.35
S7	android/telephony/TelephonyManager ;->getNetworkOperator	10.05
S4	LAUNCHER	-8.89
S5	android/os/PowerManager$WakeLock ;->release	-8.01
S2	READ_PHONE_STATE	5.03
S2	RECEIVE_SMS	-5.00
S3	c2dm.C2DMBroadcastReceiver	4.56
S2	READ_SMS	3.52
S4	DATA_SMS_RECEIVED	3.50
S5	android/app/NotificationManager ;->notify	-3.49

Sec-SVM ($\mathcal{E}_1 = 73.04\%$, $\mathcal{E}_2 = 66.24\%$, $\varepsilon_{\min} = 31$)

Set	Feature Name	r (%)
S2	READ_PHONE_STATE	3.51
S7	android/telephony/TelephonyManager ;->getNetworkOperator	3.51
S2	SEND_SMS	3.51
S3	c2dm.C2DMBroadcastReceiver	3.51
S2	INTERNET	3.44
S3	com.software.application.ShowLink	3.39
S3	com.software.application.Main	3.39
S3	com.software.application.Notificator	3.39
S3	com.software.application.Checker	3.39
S3	com.software.application.OffertActivity	3.39

SVM-RBF ($\mathcal{E}_1 = 60.74\%$, $\mathcal{E}_2 = 25.84\%$, $\varepsilon_{\min} = 31$)

Set	Feature Name	r (%)
S4	LAUNCHER	-1.89
S7	android/net/Uri;->fromFile	1.34
S5	android/os/PowerManager$WakeLock ;->release	-1.25
S2	INSTALL_SHORTCUT	1.23
S7	android/telephony/SmsMessage ;->getDisplayMessageBody	-1.21
S7	android/telephony/SmsMessage ;->getTimestampMillis	-1.20
S2	SET_ORIENTATION	-1.20
S2	ACCESS_WIFI_STATE	1.15
S4	BOOT_COMPLETED	1.08
S5	android/media/MediaPlayer;->start	-1.06

Sec-SVM ($\mathcal{E}_1 = 63.14\%$, $\mathcal{E}_2 = 52.70\%$, $\varepsilon_{\min} = 39$)

Set	Feature Name	r (%)
S2	ACCESS_NETWORK_STATE	0.93
S2	READ_PHONE_STATE	0.93
S6	READ_HISTORY_BOOKMARKS	0.93
S7	android/telephony/TelephonyManager ;->getNetworkOperatorName	-0.93
S6	ACCESS_NETWORK_STATE	-0.93
S7	android/telephony/SmsMessage;- >getDisplayOriginatingAddress	0.93
S7	android/telephony/TelephonyManager ;->getNetworkOperator	0.93
S7	android/net/Uri;->getEncodedPath	-0.93
S2	SET_ORIENTATION	-0.93
S7	java/lang/reflect/Method;->invoke	0.93

TABLE 5.2: Top-10 influential features and corresponding Gradient*Input relevance (%) for a malware of the `FakeInstaller` family (top) and a malware of the `Plankton` family (bottom). Notice that the minimum number of features to add ε_{min} to evade the classifiers increases with the evenness metrics $\mathcal{E}_1$ and $\mathcal{E}_2$.

reported in Table 5.2, which show higher values for Sec-SVM. Table 5.2 also shows the ε_{min} value, i.e. the minimum number of features to add to the malware to evade the classifier. We can notice how the ε_{min} parameter is strictly related to the evenness distribution, since higher values of $\mathcal{E}_1$ and $\mathcal{E}_2$ correspond to higher values of ε_{min}, i.e. a higher effort for the attacker to accomplish her goal. In particular, it is possible to identify a clear difference between the behaviour of SVM-RBF and Sec-SVM: the diversity of their evenness metrics, which cause the ε_{min} values to be quite different as well, indicates that, for this prediction, SVM-RBF is quite susceptible to a possible attack compared to Sec-SVM.

Conversely, considering the second sample, the attributions (regardless of the sign) and the evenness metrics present similar values. Such behaviour is also reflected in the associated ε_{min} values. In this case, the relevance values are more evenly distributed, which indicates that the evasion is more difficult.

We now correlate the evenness metrics with the *adversarial robustness* $\mathcal{R}$, introduced in Section 5.1.3.1. Figure 5.4 shows the relationship between this value and the evenness metrics for 100 samples chosen from the test set, reported for each explainability technique. From this broader view, we can see how the evenness values calculated on top of the Gradient*Input and Integrated Gradients explanations present a significant connection to the adversarial robustness. This seems not to be applicable to the Gradient technique, and specifically against the linear classifiers, whose dots in Figure 5.4 are perfectly vertical-aligned. This fact is caused by the constant value of the gradient across all the samples, which implies constant values for the evenness metrics as well. In order to assess the statistical significance of this

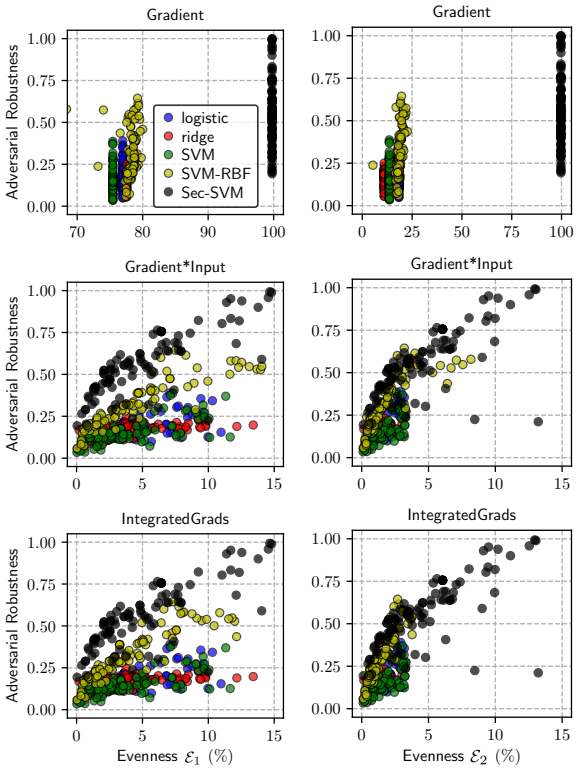


FIGURE 5.4: Evaluation of the adversarial robustness $\mathcal{R}$ against the evenness $\mathcal{E}_1$ (left), $\mathcal{E}_2$ (right) metrics for the different gradient-based explanation techniques computed on 1000 samples of the test set (only 100 samples are shown).

plot, we also compute the associated correlation values with three different metrics: Pearson (P), Spearman Rank (S), Kendall's Tau (K). They are shown in Table 5.3.

Finally, we inquire whether the connection between the evenness metrics and the detection performance of a classifier can provide a global assessment of its robustness. Figure 5.5 shows the correlation between the explanation evenness and the mean detection rate under attack, calculated for ε in the range $[1, 50]$. Similarly to the previous test, Gradient*Input and Integrated Gradients explanations present a significant connection to the adversarial robustness in most cases, while the Gradient technique does to a less extent.

5.1.5 Conclusions

With this work, we have empirically evaluated the correlation between multiple gradient-based explanation techniques and the *adversarial robustness* of different linear and non-linear classifiers against sparse evasion attacks. To this end, we have employed two synthetic measures of the *explanation evenness*, whose main advantage is not requiring any computationally-expensive attack simulations. Hence, they may be used by system designers and engineers to choose, among a plethora of different models, the one that is most resilient against sparse attacks.

As we have validated the proposed synthetic vulnerability measure by considering only the Drebin malware detector as a case study, it would be interesting to inspect other malware detectors as well as other application domains. Moreover, as the proposed metrics may be used to estimate the robustness only against sparse evasion

		Gradient		Gradient*Input		Integrated Gradients	
		$\mathcal{E}_1$	$\mathcal{E}_2$	$\mathcal{E}_1$	$\mathcal{E}_2$	$\mathcal{E}_1$	$\mathcal{E}_2$
logistic	P			0.67, <1e-5	0.75, <1e-5	0.67, <1e-5	0.75, <1e-5
	S			0.67, <1e-5	0.72, <1e-5	0.67, <1e-5	0.72, <1e-5
	K			0.51, <1e-5	0.54, <1e-5	0.51, <1e-5	0.54, <1e-5
ridge	P			0.48, <1e-5	0.56, <1e-5	0.48, <1e-5	0.56, <1e-5
	S			0.58, <1e-5	0.67, <1e-5	0.58, <1e-5	0.67, <1e-5
	K			0.41, <1e-5	0.49, <1e-5	0.41, <1e-5	0.49, <1e-5
SVM	P			0.68, <1e-5	0.70, <1e-5	0.68, <1e-5	0.70, <1e-5
	S			0.66, <1e-5	0.73, <1e-5	0.66, <1e-5	0.73, <1e-5
	K			0.49, <1e-5	0.54, <1e-5	0.49, <1e-5	0.54, <1e-5
SVM-RBF	P	0.03, 0.769	0.46, <1e-5	0.82, <1e-5	0.82, <1e-5	0.89, <1e-5	0.91, <1e-5
	S	0.46, <1e-5	0.70, <1e-5	0.94, <1e-5	0.94, <1e-5	0.93, <1e-5	0.93, <1e-5
	K	0.34, <1e-5	0.51, <1e-5	0.81, <1e-5	0.80, <1e-5	0.78, <1e-5	0.77, <1e-5
Sec-SVM	P			0.73, <1e-5	0.76, <1e-5	0.73, <1e-5	0.76, <1e-5
	S			0.76, <1e-5	0.78, <1e-5	0.76, <1e-5	0.78, <1e-5
	K			0.62, <1e-5	0.67, <1e-5	0.62, <1e-5	0.67, <1e-5

TABLE 5.3: Correlation between the adversarial robustness $\mathcal{R}$ and the evenness metrics $\mathcal{E}_1$ and $\mathcal{E}_2$. Pearson (P), Spearman Rank (S), Kendall's Tau (K) coefficients along with corresponding p-values. The linear classifiers lack a correlation value since the evenness is constant (being the gradient constant as well); therefore, resulting in a not defined correlation.

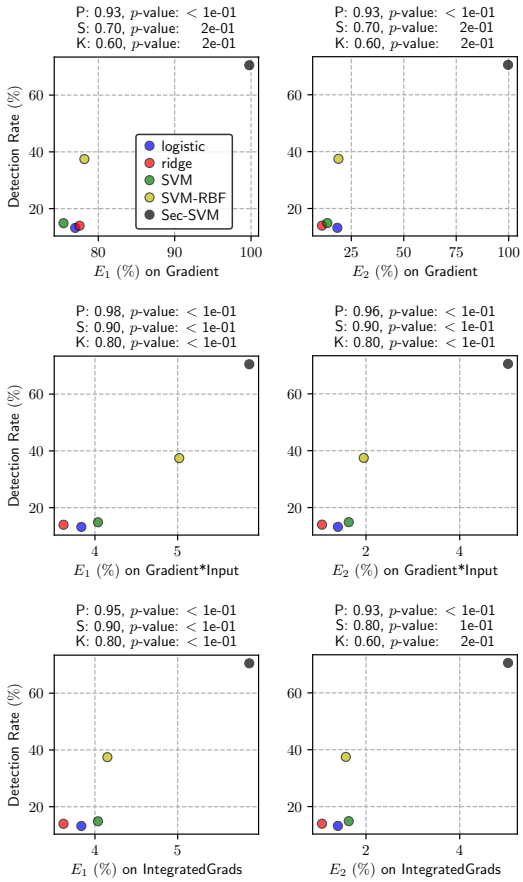


FIGURE 5.5: Evaluation of the evenness metrics E_1 (left) and E_2 (right) against the Detection Rate (FPR 1%) for the different gradient-based explanation techniques computed on the Drebin dataset.

attacks, another research direction could be to devise a similar measure robustness estimation when the attack is subjected to different application constraints. Also, it could be interesting to assess if our vulnerability measures can be successfully applied when the attacker does not know the classifier parameters or when the model is not differentiable; in that case, a surrogate classifier would be used to explain the original unknown model function.

Finally, another research avenue is to modify the objective functions used to train the considered machine learning models by adding to them a penalty which is inversely proportional to the proposed evenness metrics, in order to enforce the classifier to learn more evenly-distributed relevance scores and, consequently, the model robustness.

5.2 Evaluating the Feasibility of Adversarial Sample Creation

After presenting methods to assess Android detectors' vulnerabilities *in vitro*, in the following, I illustrate the problem of understanding the possibility of fulfilling evasion attacks *in vivo*. As a matter of fact, a critical problem that has been often overlooked in previous work is the practical feasibility of generating adversarial samples.

When designing a machine learning-based detection system, experts identify a set of characteristics (features) that are expected to be effective in classifying the input samples. That is, they create a feature space where they *map* specific characteristics, patterns, or behaviours of the sample to a feature vector. Conversely, when performing adversarial attacks, attackers generate an altered feature vector. Thus, converting each alteration into a modification of the samples (in the *problem space*) in order to attain the desired evasion. Depending on the setting, this transition is not straightforward and entails the difficulty of moving from the feature space to the problem space (or vice versa), i.e. finding an exact, unique correspondence between values in the feature vector and characteristics in the problem domain (e.g. functionalities in an Android application). This is the so-called *inverse feature-mapping problem* [27, 81, 102] — or the more generic problem-feature space dilemma [103]. Moreover, the generation of concrete, realistic adversarial samples also requires taking into account different constraints, such as preserving the app's semantics or keeping it plausible for human inspection [104, 105].

In this work by Cara et al. [93], we have explored the development of evasive Android apps in the problem space. In particular, we probe the feasibility of generating such samples specifically through the injection of system API calls (the same kind of features of Chapter 4). We analyse an injection strategy that only adds the calls needed to achieve the evasion by preserving the application's overall functionality. Consequently, we consider a scenario where both the attacker's effort and the impact on the generated app are kept at a minimum level. Moreover, differently from all previous articles in the literature, we evaluate our strategy on a detector designed with non-binary features. Overall, the key elements of this work are the following:

1. We discuss the constraints required to create concrete, working Android adversarial samples through API call injection;

2. We evaluate the feasibility of injecting system API calls by both identifying the subset of the usable ones and explaining their relevance to evasion through a gradient-based interpretability technique;

3. We evaluate the effectiveness of mimicry and random noise addition attacks against a state-of-the-art ransomware detector that employs non-binary features;

4. We develop a basic implementation of the considered injection strategy that creates working adversarial malicious samples.

This way, we believe this proposal starts to highlight the actual, viable injection strategies attackers have at their disposal for the creation of evasive Android malware samples.

In the following, the threat model of our setting is depicted (Section 5.2.1), followed by a discussion about the problem space and its constraints (Section 5.1.2). Our implementation for the generation of Android adversarial samples is described in Section 5.2.3, while the experimental analysis is presented in Section 5.2.4. Section 5.2.6 makes conclusive remarks.

5.2.1 Threat Model

Using the same schema of Section 2.1, in the following, the threat model of this work is illustrated.

Attacker's goal In our setting, the attackers aim to make the detection system classify ransomware apps as trusted ones. In particular, since we consider the *evasion* strategy, they fulfil this goal by modifying the Android apps at test time rather than by poisoning the system's training set.

Attacker's knowledge Attackers often have incomplete knowledge about the target system. For this reason, in this work, we simulate a scenario where the attacker has minimum information about the Android detection system. Specifically, we focus on the *mimicry attack*, which has been studied in previous work [17, 27, 102]. In this case, $\boldsymbol{\theta} = (\hat{D}, X)$, which means that the attacker knows the feature space and owns a set of data that is a representative approximation of the probability distribution of the data employed in the target system. This is a more realistic scenario, in which the attacker can modify the features of a malicious sample to make its feature vector as similar as possible to one of the benign samples at its disposal. As a comparison, we also consider a *random noise addition attack*, which does not allow targeting a specific class, but can be useful to provide a generic assessment of the vulnerability of the system to perturbed inputs.

Attacker's capability The Android application architecture mostly allows the attacker to only add new elements to the app (*feature addition*), such as permissions, strings, or function calls, but does not permit removing them (*feature removal*). I discuss this aspect in more detail in the section that follows.

5.2.2 The Problem Space Domain

The goal of this work is to evaluate to what extent it is feasible to generate real-world Android adversarial samples. In particular, the focus of our analysis lies specifically on the constraints and consequences of injecting system API calls. With this respect, the first concern to consider is the so-called *inverse feature-mapping* problem — or the more generic *problem-feature space dilemma* [103]. As hinted in Section 5.2, this refers to the difficulty of moving from the feature space to the problem domain (or vice versa), i.e. finding an exact, unique correspondence between values in the feature vector and characteristics in the problem domain, e.g. functionalities in an Android application. The feature mapping problem can be defined as a function ψ that, given a sample z, generates a d-dimensional feature vector $\boldsymbol{x} = [x_1, x_2, ..., x_d]$, such that $\psi(z) = \boldsymbol{x}$ [104]. Conversely, the opposite flux in the inverse feature-mapping case is a function $\mathcal{S}$, such that taking a feature vector $\boldsymbol{x}$, we have $\mathcal{S}(\boldsymbol{x}) = z'$. However, it is not guaranteed that $z \equiv z'$.

As an example, let us consider the feature vector of our setting, which consists of the occurrence of API package calls. Due to this choice, the value of a feature x_i in the feature vector can be increased through two main behaviours in the Android application: *(i)* the call of a class constructor and *(ii)* the call of a method. In both cases, the class involved in these calls must belong to the package that corresponds to the i-th feature. This means that there are as many ways to change that feature value as the number of callable classes in the package. Figure 5.6 exemplifies this aspect by showing, for a few packages, their related classes that we identify as callable for our purposes. By contrast, an alternative feature vector (as discussed in Section 4.2) that describes the occurrence of system API method calls would have a one-to-one mapping between the i-th feature and the call of the corresponding method.

android.content -> ContentUris, ContentValues, Intent, IntentFilter
android.database -> ContentObservable, DatabaseUtils, DataSetObservable
android. graphics -> Picture, Regiorn, ColorFilter, Camera

FIGURE 5.6: Example of callable classes for three different API packages.

The above-described issue is particularly relevant for the creation process of adversarial samples. Another implication to consider is the potential presence of *side-effect features*, i.e. the undesired alteration of features besides the ones targeted in the attack [104]. For example, inserting whole portions of code to add specific calls may have the effect of injecting unnecessary, additional calls. This may lead to an evasive feature vector that is slightly different from the expected one; therefore, making the behaviour of the target classifier unpredictable.
The injection approach considered in this work starts from the will of inserting the minimum amount of modifications needed to evade the detection. However, other concerns must be taken into account in order to create a realistic, working adversarial malware. I discuss them below.

5.2.2.1 Constraints

I now present the constraints that we consider in our approach and their implications on the injection strategy design. I illustrate them on top of the definitions proposed by Pierazzi et al. [104]. I refer the reader to that work for further details.

Available transformations The modification of the features has to correspond to doable actions in the problem domain. That is, it is necessary to evaluate the set of possible transformations. In the case of Android, some sample modifications could lead to a change in the app behaviour, a crash during the execution, or rejection by the Android Verifier. Typically, the attacker can only add new elements to the apps (*feature addition*), such as permissions, strings, or function calls, while it is harder for it to remove them (*feature removal*). For example, it is not possible to remove permissions from the `manifest`.

In this work, we choose the feature addition strategy, as we only inject new system API calls into the `dex` code. In this sense, it is possible to successfully perform the modifications only for a reduced set of Android system packages and classes. In fact, the packages we inject are the ones whose classes are not an `interface` or `abstract` classes and whose constructors are public and accessible. Another issue is related to the call parameters. These have to be correctly defined because Java has a strict, static type checking. Hence, to call methods or constructors that receive specific parameters, one could create and pass new objects of the needed classes. Since this can result in being a complicated procedure, in this work, we start exploring the most straightforward setting for attackers, i.e. where they restrict the set of callable classes to the ones that need primitive or no parameters at all. We evaluate both cases in Section 5.2.4.2, and we implement the no-parameters case.

Preserved semantics The transformations must preserve the functionality and behaviour of the original sample, e.g. the malicious behaviour of Android malware. To check if the application's behaviour has been kept unchanged after the injection, one could build a suite of automatic tests to perform basic operations. For instance, it is possible to open and close the main `activity`, put it in the background, then verify if the produced output is the same as the one of the original app. In our setting, the main criticality of the injection of API calls is related to the execution of operations that could lead to crashes or block the execution, which is especially relevant when calling methods, while more manageable when calling only class constructors. More specifically, a call may require a reference to non-existent objects, causing an exception in the execution (e.g. `openOptionsMenu()` from `android.view.View` if no Option Menu is present) or may block the user interface if it runs in the main thread.

Plausibility The created adversarial samples have to be *plausible* on human inspection, i.e. they should not contain evident (from a human perspective) signs of manipulation. For example, having 50 consecutive calls of the same method inside the same function would be extremely suspicious. However, this concept is also quite tricky in practice; in fact, there are no general and automatic approaches to evaluate it. In our work, we pursue this constraint by limiting the repetition of the same calls multiple times in the same portion of the app. In particular, we spread the injected calls throughout the whole app in order to make a sequence of constructor calls less

likely. However, a more sophisticated strategy should take care of the coherence of the injected code with the application context. For instance, adding permissions that do not pertain to the app's scope could be suspicious to the human expert.

Robustness The alterations made to the samples should be resilient to preprocessing. For example, injecting dead code in the app is common for attackers, but it is easy to neutralise through dead code removal tools. In this sense, our approach aims at the injection of code that is properly executed.

5.2.2.2 API Injection Feasibility

In the specific setting of this work, successfully creating the adversarial samples implies carefully selecting the system APIs to inject. Therefore, to implement the attack, the first step is to identify what API constructors are usable. Starting from the complete set of them for each package of each API level, we remove the ones that *(i)* are not public or have protected access, *(ii)* belong to abstract classes, *(iii)* potentially throw exceptions (hence, requiring a more complex code injection), and *(iv)* receive parameters of non-primitive types. Then, we identify as usable the classes that have at least one constructor satisfying this filtering; consequently, we also derive the packages that have at least one class available. Moreover, we consider two cases on the input parameters of the constructors: the first one — which we call 'no-parameters' — computes the values on the basis of the constructors that receive no parameters in order to be called. In the second case — which we call 'primitive-parameters' — we also include constructors that receive parameters of primitive types, where *primitive* indicates one belonging to the following types: `int`, `short`, `long`, `float`, `double`, `char`, and `boolean`. Notably, attackers could include other non-primitive types that are simple to manage, such as `java.lang.String`. In Section 5.2.4.2, we evaluate the attained results quantitatively.

Explaining evasion Besides identifying the modifiable packages and classes at the disposal of the attacker, it would be interesting to understand if and to what extent the usable APIs turn out to be the ones that are effective for evasion attacks; that is, the ones that, when modified, move the adversarial sample towards the benign class.

To perform this kind of evaluation, we make use of Integrated Gradients (see Section 2.2.1). As the work illustrated in Section 5.1 has shown, it is possible to put a bridge between gradient-based explainability techniques and evasion attacks specifically in the Android malware detection domain. Therefore, the rationale behind using explanations in this case is that we expect that a relevant feature is significant for a successful realisation of the evasion attack.
Since the attribution values of Integrated Gradients can be calculated with respect to a specific output class of the classifier, we consider the trusted one. This way, positive values indicate that a feature moves the prediction towards the trusted class. Consequently, *we consider a feature as relevant (for evasion) when its attribution value is strictly positive.* In other words, we identify the features that influence the classification in the direction of the trusted class and, consequently, the ones that an attacker should modify to evade the detection of the considered sample. I show this assessment in Section 5.2.4.2.

5.2.3 Adversarial Malware Creation

The core of our implementation is based on two libraries: DexLib [106] and Apktool [107]. Figure 5.7 shows the architecture of our implementation of the system to generate adversarial samples according to a mimicry attack—or a random noise one alternatively. The system takes as input a malicious Android sample and gives as output an adversarial malware that is expected to be classified as benign. The system chain consists of three phases that I describe in the following.

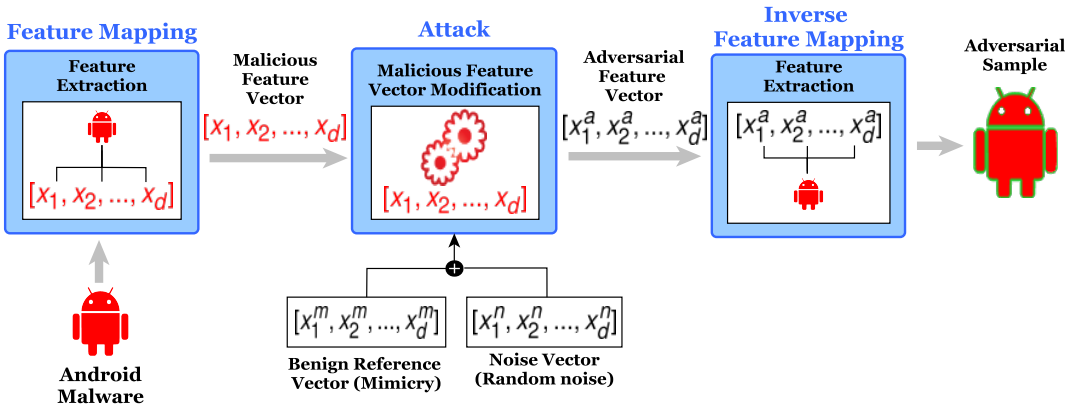


FIGURE 5.7: The architecture of the system. Firstly, it processes the given malicious sample to retrieve its feature vector. Then, it performs the modifications to the feature vector using either the benign reference vector (mimicry) or the noise vector (random noise adding). Finally, it generates the adversarial malicious sample.

Feature Mapping In this phase, the malicious Android sample is statically analysed to detect and count all the Android system API calls and create the numeric vector of features $\boldsymbol{x} = [x_1, x_2, ..., x_d]$. As shown in Figure 5.8, firstly, we parse the `.dex` files (using Dexlib) to get the API called through the `invoke` functions (see Chapter 3), matching the Android system calls previously gathered from the official Android documentation. Then, we count the occurrences for each API call. Finally, we generate a sparse feature vector where, for every non-zero feature, there is the occurrence of the references to the specific package inside the analysed Android application.

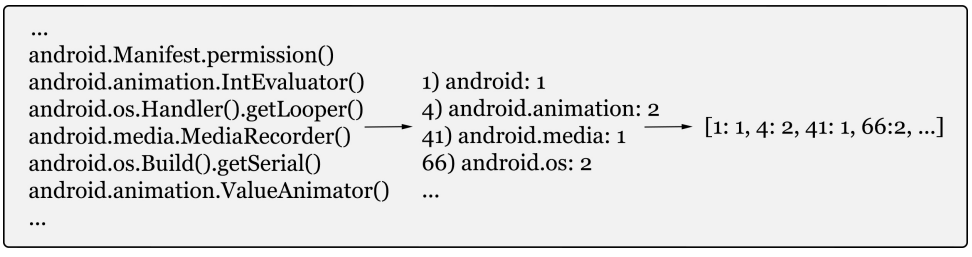


FIGURE 5.8: Example of feature mapping for the creation of the feature vector.

Attack The extracted malicious feature vector is then modified to perform the attack. This phase aims to generate an adversarial feature vector using a mimicry or a random noise addition approach. As shown in Figure 5.7, for the mimicry case,

we choose as a reference a unique benign feature vector $\boldsymbol{x}^m = [x^m{}_1, x^m{}_2, ..., x^m{}_d]$. The choice of this vector can be made in different ways. Specifically, one might choose the benign reference vector to be added to the malicious sample according to different strategies. In Section 5.2.4.3, we compare the results of our experiments for four ways of choice, which we call: 'mean', 'median', 'real mean', and 'real median'. Basically, in the first two cases, we take as the reference vector the mean (median) vector among the trusted samples available to the attacker, i.e. the test set; in the remaining two cases, we consider the real sample of the test set that is closest to the mean (median) vector. Specifically, we take the real sample with the highest cosine similarity, calculated with respect to the reference feature vector through the following formulation:

$$\text{Cosine Similarity}(\boldsymbol{x}, \boldsymbol{x^m}) := \frac{\sum_{i=1}^{d} x_i x^m{}_i}{\|\boldsymbol{x}\| \|\boldsymbol{x}^m\|} \tag{5.11}$$

A noise vector $\boldsymbol{x}^n = [x^n{}_1, x^n{}_2, ..., x^n{}_d]$ is instead added in the case of random noise addition. We consider different implementation strategies for the attack (see Section 5.2.4.3) also in this case. We define these strategies 'absolute' and 'relative'. The first one consists of increasing, for each feature, the occurrence of the correspondent call with a randomly chosen value between zero and the considered noise level (e.g. 10). In the second one, the features are added by taking into account the original value of each feature in the sample. For example, with an original feature value of 20 and a noise level of 50%, we randomly increase the occurrence with a value between 0 and 10.

Once we obtain the vector that enables the modification, for the mimicry case, we compute the difference between the reference feature vector and each malicious one, then add the resulting features to the malicious sample, creating the adversarial feature vector $\boldsymbol{x}^a = [x^a{}_1, x^a{}_2, ..., x^a{}_d]$. Notably, if the original malicious sample exhibits one or more features with values higher than those of the reference vector, we keep the same value of the original sample (otherwise, we would have performed feature removal). Regarding the random noise addition case, the noise vector is added to the malicious feature vector to create the adversarial feature vector $\boldsymbol{x}^a$.

Inverse Feature Mapping This phase is the opposite of the feature mapping phase, so each value of the adversarial feature vector, which is not already in the malicious sample, is matched with the corresponding Android system call and added in the malicious sample. We use Apktool to disassemble the Android application; then, we employ Dexlib to perform all the modifications on the bytecode level. Finally, we leverage Apktool again to reassemble and sign the generated adversarial sample, which is manually tested to verify that the functionality is preserved. As introduced in Section 5.2.2, for each feature (package), there may be more than one usable constructor since multiple classes can be available. Thus, for each feature, we randomly choose a constructor among the available ones. In this way, we increase the plausibility of the adversarial app, as it would be easier for a code analyst to notice the same class called multiple times rather than different classes of the same package. In this sense, we also spread the injected calls across all the methods already defined in the `.dex` file, so that they are not concentrated in a unique portion of code.

Each injected call is added by defining an object of the related class and calling its constructor. Figure 5.9 shows a smali-like representation of the injection code.

We can see a new-instance function to create an object of the class to add and an invoke-direct function to call the constructor of that class. The injected instructions are placed before the `return` statement of the selected method. Notably, the choice of calling constructors does not cause any side-effect since no features other than the target ones are modified within the feature vector.

```
new-instance v0, java.util.concurrent.atomic.AtomicLong;
invoke-direct v0, java.util.concurrent.atomic.AtomicLong;-><init>()V
```

FIGURE 5.9: Example of the injection of an Android system call.

5.2.4 Experimental Results

In this section, after detailing our experimental setup (Section 5.2.4.1), we evaluate the capability for the attacker to inject Android's system API calls (Section 5.2.4.2). Then, I show the performance of the mimicry and the random noise attacks (Section 5.2.4.3), as well as their impact in terms of added calls with respect to the original samples (Section 5.2.4.4).

5.2.4.1 Setting

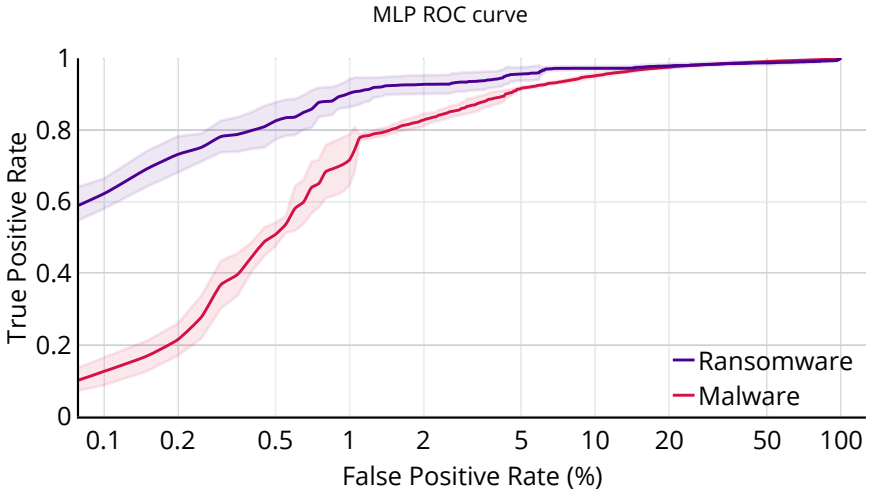


FIGURE 5.10: Average ROC curve of the MLP classifier over repeated 5-fold cross-validation. The lines for the ransomware and malware classes include the standard deviation in translucent colour.

Dataset We use the same dataset as [18] (and Section 4.2), composed of 39157 Android applications, 3017 of which are ransomware applications, 18396 trusted applications, and 17744 generic malware applications.

Features We use R-PackDroid (see [18, 76] and Chapter 4) as a reference detection system. The feature set consists of the cumulative list of system API packages up to Level 26 (Android Oreo), for a total of 196 features. In particular, the set of APIs is strictly limited to the Android *platform* [86] ones.

Classifier We have trained an MLP (multilayer perceptron) neural network with Keras. We have performed a repeated five-fold cross-validation, along with a search for the best hyperparameters (e.g. number of layers, number of neurons per layer) for the net. Figure 5.10 shows the mean ROC curve over the five repetitions.
It is worth recalling that, as explained in Section 4.3.2.1, the detection performance of MLP is not the optimal one for this setting since, in our tests, we had attained better results with a Random Forest algorithm. However, explanations with Integrated Gradients cannot be produced from Random Forest due to the non-differentiability of its decision function. Therefore, to keep our setting coherent, we have chosen to perform all the experiments on the MLP classifier.

5.2.4.2 API Injection Evaluation

As explained in Section 5.2.2, attackers have to take into account the feasibility of the alterations of their malicious samples. Table 5.4 shows, for each API level up to 29, the percentage of usable packages and classes out of the whole set of APIs. In particular, Table 5.4a is related to the *no-parameters* case. Depending on the Android API level, it is possible to cover several packages, from 51% to 57%, and several classes between 15% and 27%.

In the second case — *primitive-parameters* — of Table 5.4b, we also include constructors that receive parameters of primitive types. With this variation, it is possible to cover more packages, between 55% and 61%, and more classes, between 18% and 33%, depending on the Android API level. Overall, we point out that the results are similar in both cases, but the effort in the first case is lower since that attacker does not need to create and inject new variables with the correct primitive type. Therefore, the first approach is more convenient for an attacker that wants to generate adversarial samples.

It is worth noting that the attacker's goal is to infect the highest number of devices possible. Consequently, the minimum API level of a malicious sample tends to be very low. For example, by extracting the `sdkversion` field in the `Manifest` of each ransomware sample of our test set, we have verified that several apps lie in the 7–9 range. Therefore, attackers are encouraged to inject older APIs rather than the newer ones.

TABLE 5.4: Number of available packages and classes for each Android API level. In (a), only constructors without parameters are considered, while in (b), we also include constructors that receive primitive parameters.

(a)

Granularity (%)	**Android API Level**																												
	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
Packages	53	54	55	56	56	56	55	55	55	56	57	57	56	56	55	55	55	55	55	54	53	53	53	53	51	51	51	51
Classes	27	26	26	25	25	25	24	24	24	24	24	24	23	23	22	22	21	21	21	19	19	18	18	18	17	17	16	15

(b)

Granularity (%)	**Android API Level**																												
	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
Packages	58	59	59	61	61	61	60	59	59	60	61	61	60	60	59	59	58	58	58	57	58	58	58	58	56	56	55	55
Classes	33	32	32	31	31	31	30	29	29	29	29	29	28	28	27	26	26	25	25	23	23	22	21	21	20	20	19	18

Are the modifiable features effective for evasion attacks? The number of available packages and classes inferred in the previous experiment appears to be not really high. However, as introduced in Section 5.2.2.2, it is also worth inspecting the importance of the usable features to the classification. From now on, we conduct all of our experiments using the ransomware samples of the test set, which emulates the set of samples at the attacker's disposal. We compute the explanations using DeepExplain [88].

As a first point, we evaluate the percentage of relevant features modifiable by the attacker for each sample. We attain a mean value across the samples of 72.1% for the no-parameters case and of 73.8% in the primitive-parameters one. This result suggests that the attacker can modify a good number of useful features to evade detection. As a second test, we identify which relevant features are the most frequent among the modifiable ones. The results are shown in Figure 5.11. As can be seen, the shown features correspond, as expected, to the ones that are known to be descriptive of the trusted samples, i.e. a broad set of functionalities related, for example, to the app's user interface.

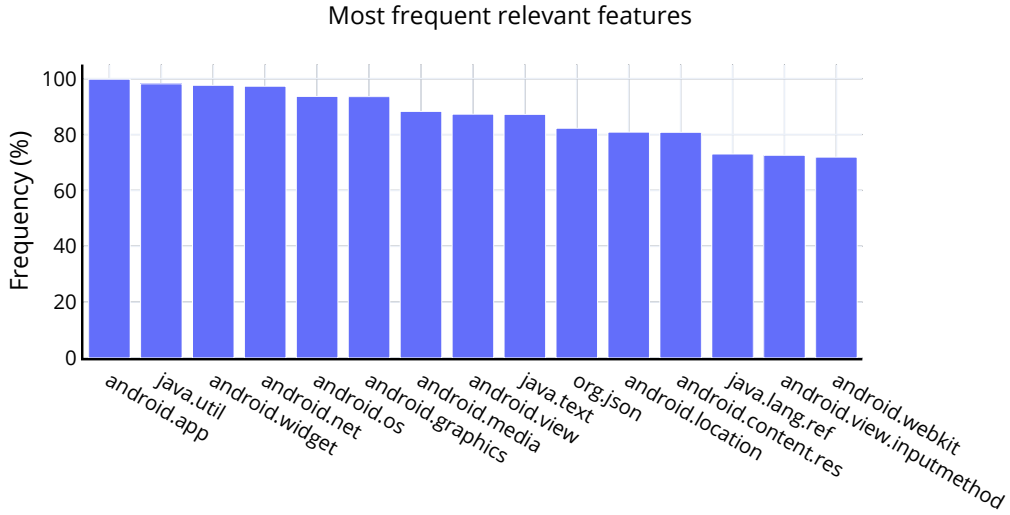


FIGURE 5.11: Top 15 relevant features among the usable ones.

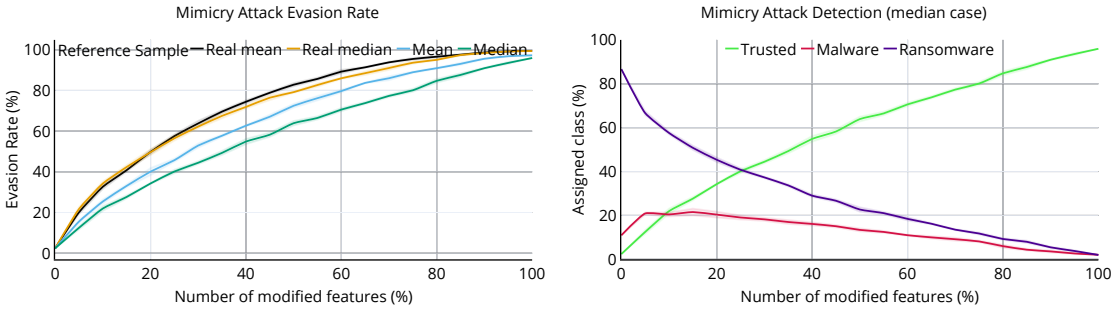
5.2.4.3 Attack Results

In the following, we assess the mimicry attack's performance and, as a comparison, the most straightforward attack possible, the random noise one.

Mimicry Attack Figure 5.12a shows the evasion rate for four different reference samples, as illustrated in Section 5.2.3; that is, how many adversarial samples evade the classification. In the x-axis, we evaluate an increasing percentage of modified features, i.e. the number of injected packages out of the total number of modifiable ones by the attacker. Since the choice of the subset of features to modify for each percentage level below 100% is random, we show the average result over five repetitions. Moreover, it is worth noting that each chosen feature exhibits a different number of calls to inject because some APIs are being called in the app thousands

of times, while others might be referenced only a few times. In the same figure, it is possible to see that all the curves present similar trends.

In Figure 5.12b, we focus on the median strategy and show how the samples are classified. We can see that the evasion works as desired because the number of samples classified as trusted increases while the number of samples classified as ransomware and malware decreases. Notably, in Figure 5.12a, the evasion rate at 0% of modified features is not zero because some ransomware samples are mistakenly classified as benign or malware. For the same reason, Figure 5.12b shows that the detection of the samples as ransomware is not 100% when no features are modified.

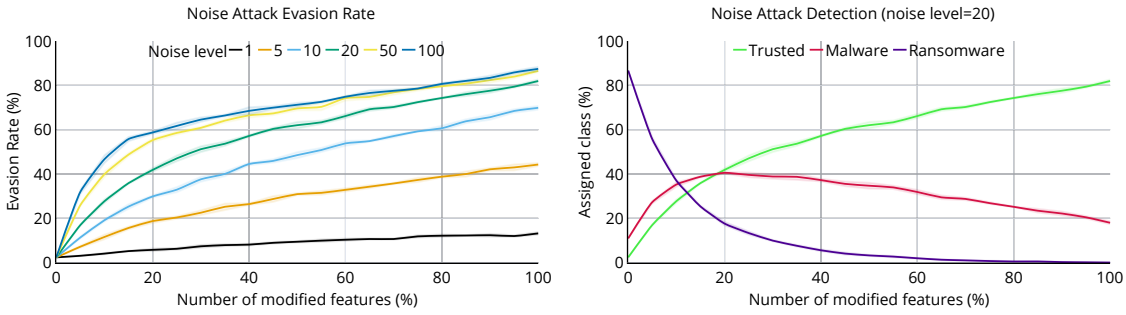


(a) (b)

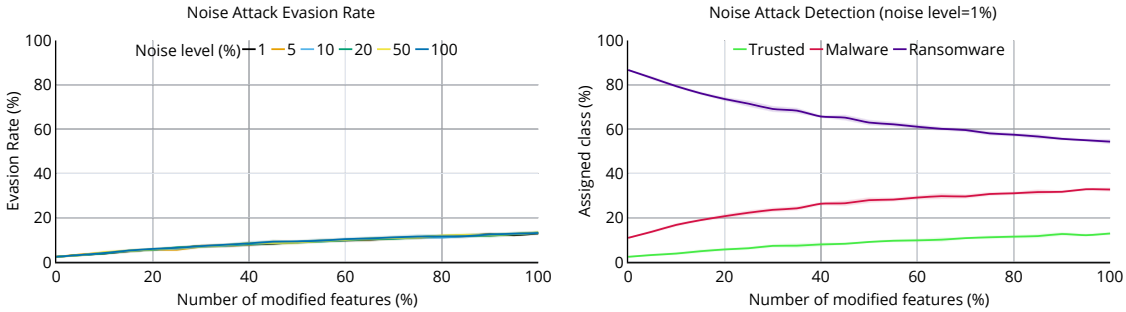
FIGURE 5.12: Mimicry attack's evasion rate for different choices of the reference sample (a). Detection rate detail of the ransomware samples (b). Both figures are the average results over five repetitions and include the standard deviation in translucent colour.

Random Noise Attack After the mimicry attack, we have performed a random noise addition attack, which can be useful as a reference since it only consists of injecting API calls without any specific pattern or target class. Following the same procedure as Section 5.2.4.3, the results are shown in Figure 5.13. Specifically, the absolute approach is the one shown in Figure 5.13a. On the left side, we evaluate the evasion rate for different levels of added noise. As we can see, the higher the noise level is, the higher the evasion rate is. On the right side of the figure, we show the detail of the assigned classes for a noise level equal to 20, which achieves similar evasion levels as the mimicry case. As we can see, the curve also exhibits the same tendency, i.e. it causes an increasing detection of the ransomware samples as legitimate ones as if it was a targeted attack. This is significant since it seems to suggest that no specific injection pattern is needed to make ransomware samples classified as trusted ones. Consequently, attackers would not need a set of trusted samples with the same probability distribution of the target system's training set, which is necessary to perform the mimicry attack.

The relative approach is shown in Figure 5.13b. The evasion rate on the left side of the figure is, in this case, completely different from the absolute one, reaching a value of around 15% at the highest point. Notably, there is no significant difference between each noise level. This can be related to the sparsity of the samples' feature vector. In fact, several features have a zero value, so the percentage of a zero value would always end up with no addition. Therefore, in our implementation, we chose to set a random increase between zero and one. Ultimately, this result shows that adding a high percentage of noise only to the features that already had non-zero values is insufficient to accomplish the evasion.



(a) Absolute noise addition



(b) Relative noise addition

FIGURE 5.13: Evasion rate for different levels of noise injection in the absolute and relative cases (left side of the figure). Detection detail for a noise level equal to 20 (right side of (a)) and a 1% noise level (right side of (b)). All the results are the average over five repetitions and include the standard deviation in translucent colour.

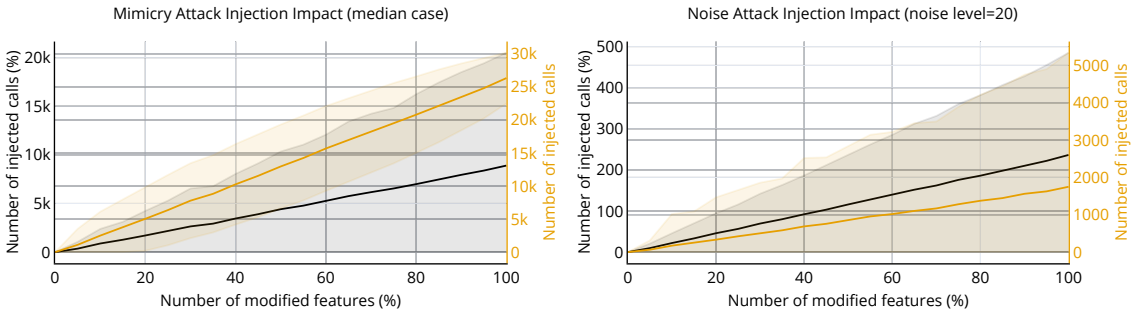
5.2.4.4 Injection Impact

The mimicry and the random noise addition attack results have shown that the evasion rates can go up to around 80%. This appears to be an outstanding result; however, it does not depict the full view of the problem. For example, it does not tell anything about the plausibility of the adversarial sample. In fact, we may say that the plausibility for the method proposed in this work is inversely correlated to the number of injected features. The more additional calls (with respect to the original sample) there are, the lower is the plausibility. Therefore, with Figure 5.14, we evaluate the impact on the samples of both the considered attacks (median case for the mimicry attack, absolute noise level equal to 20 for the noise attack), in terms of extra calls added. As with the previous figures, in the x-axis, we put an increasing percentage of modified features. We insert two y-axes to show the average number of added calls both as a percentage of the original amount (left axis) and as the absolute number (right axis). As can be seen in Figure 5.14a, the mimicry attack causes a massive increase in calls. For example, let us fix the evasion rate to 50%, which requires modifying around 35% of the modifiable features (see Figure 5.12a). With this setting, the increment of system API calls would be almost nine thousand per cent on average. Notably, the standard deviation is quite high, so this number can be significantly higher or lower; nevertheless, the generated adversarial sample would most likely not be realistic.

The random noise addition attack attains much better results in this sense. A 50% evasion rate is accomplished by modifying around 30% of the modifiable

features (see Figure 5.13a), which means injecting 69% additional calls with respect to the original samples on average. Although the difference with the mimicry attack is significant, the level of additional calls to add results in being too high to be plausible.

Overall, the results show the detector's vulnerability to perturbed inputs, even with a non-targeted attack. However, to achieve a sufficient evasion rate, attackers need to inject a vast number of calls, which weakens the attack's plausibility.



(a) (b)

FIGURE 5.14: Average impact on the number of calls for mimicry (a) and noise (b) attacks. The standard deviation is also reported in translucent colour.

5.2.5 Related Work

While it is possible to find multiple research efforts about the formal aspects of the evasion attack, few of them have focused on creating adversarial samples that accomplish this kind of attack in practice.

Concerning the Android domain, Pierazzi et al. [104] is currently the most comprehensive work in the literature. The authors have proposed a formalisation for problem space attacks and a method to create evasive Android malware. In particular, starting from the feature space problem, which has been widely discussed in the literature, they have identified the constraints (the ones discussed in Section 5.2.2.1) to keep the generated samples working and realistic. This means creating adversarial samples that are *(i)* robust to preprocessing analysis, *(ii)* preserved in their semantics, *(iii)* completely functioning, and *(iv)* feasible in its transformations. To generate the adversarial samples, they have used an automated software transplantation technique, which consists of taking a piece of code (that contains the desired features) from another application. Finally, they have evaluated their system on Drebin [15] and its hardened variant, which uses a Sec-SVM classifier (see Section 5.1.2.2) [17]. They have shown that it is possible to create a sample with high evasion rate by making about two dozen transformations for the SVM classifier and about a hundred transformations for the Sec-SVM classifier. While this work is robust and effective in its methodology, it could be possible to use an opaque predicate detection mechanism to detect the unreachable branches [108, 109].
Grosse et al. [32] have proposed a sample generation method with only one constraint: keeping the semantics of the evasive samples consistent with a maximum of 20 transformations. They have tested their system with a deep neural network classifier trained using the Drebin dataset. However, their work has only performed

minimal modifications to the `Manifest`, so these modifications may be detected with a static analysis tool made to remove unused permissions and undeclared components. Furthermore, the adversarial samples have not been tested, so there is no way to know whether the adversarial samples were correctly executed.
Yang et al. [110] have proposed a methodology to create adversarial Android malware following two principal constraints: preserving the malicious behaviours and maintaining the robustness of the application (e.g. correct installation and execution). They have evaluated their adversarial malware system against the Drebin classifier [15] and the AppContext classifier [63]. However, their methodology lacks in preserving the stability of the application (e.g. they show high rates of adversarial application crashes), especially when the number of features to add increases.
It is worth noting that, differently from our setting (see Section 5.2.3), all the above-mentioned articles evaluate their proposals on systems with binary features; thus, only highlighting the presence or absence of certain characteristics in the app. Table 5.5 performs a comparison of our method with the above-described works.

For what concerns other domains, Song et al. [105] have presented an open-source framework to create adversarial malware samples capable of evading detection. The proposed system firstly generates the adversarial samples with random modifications; then, it minimizes the sample, removing the useless features for the classification. They used two open-source classifiers: Ember and ClamAV. They also described the interpretation of the features to give a better explanation of why the generated adversarial samples evade the classification. They have shown that the generated adversarial malicious samples can evade the classification and that, in some cases, the attacks are transferable between different detection systems.
Rosenberg et al. [113] have proposed an adversarial attack against Windows malware classifiers based on API calls and static features (e.g. printable strings). They evaluated their system with different variants of RNN (recurrent neural network) and traditional machine learning classifiers. However, their methodology is based on the injection of no-op API calls that may be easily detected with a static code analyser.
Hu and Tan [114] have proposed a generative adversarial network (GAN) to create adversarial malware samples. They have evaluated their system on different machine learning-based classifiers. However, their methodology is not reliable because of the use of GAN, which is known to have an unstable training process [115].

5.2.6 Conclusions

With this work, we have presented a study about the feasibility of performing a fine-grained injection of system API calls to Android apps in order to evade machine learning-based malware detectors. This kind of strategy can be particularly effective for creating a massive amount of adversarial samples with a relatively low effort by the attackers. However, we have discussed the necessity of satisfying several constraints to generate realistic, working samples. The experimental results show that both the mimicry and the random noise attacks, which do not require a high level of knowledge about the target system, suffice to evade classification. Nevertheless, they cause a substantial loss of plausibility of the generated adversarial sample since they add an excessive number of additional calls; therefore, weakening the effectiveness. Therefore, we believe that future work should focus on performing this kind of

TABLE 5.5: Comparison of related work on practical creation of evasive Android apps.

	This work	**Pierazzi et al. [104]**	**Yang et al. [110]**	**Grosse et al. [32]**
Reference classifier	R-Packdroid	Drebin	Drebin, AppContext [63]	Drebin (partially)
Security analysis	Static	Static	Static	Static
Modified components	Dex	Dex, Manifest	Dex, Manifest	Manifest
Feature type	Integer	Binary	Categorical (AppContext), binary (Drebin)	Binary
Trasformation	Code addition	Code addition, through automatic software transplantation	Code addition and modification, through automatic software transplantation	Code addition
Semantics	Preserved	Preserved (injected code is not executed)	Preserved for a limited amount of mutations	Preserved
Robustness to preprocessing	Not explicitly tested. Not fully robust to removal of redundant code*	Robust to: removal of redundant code, undeclared variables, unlinked resources, undefined references, name conflicts, no-op instructions	Not explicitly tested	Not robust to removal of unused permissions and undeclared components
Plausibility	Mutated apps install and start on an emulator (limited number of samples tested). Code is not realistic for the considered attack types*	Code is realistic. Mutated apps install and start on an emulator	Apps with a limited amount of mutations install, start on an emulator, and perform expected malicious code. Code is realistic	Mutated apps not tested. Additions are not realistic
Side effects	No	Yes	Yes	No
Attack type	Mimicry and random noise	Gradient-based	Gradient-based	Gradient-based
Suggested defences	Preprocessing*	Monotonic classifiers [111]	Adversarial training, known malware code detection, weight bounding (like Sec-SVM)	Adversarial training, distillation [112]

* Strongly influenced by the number of injections the attack type requires.

analysis using a gradient approach that minimises the number of injected features to preserve plausibility. This aspect is also relevant to highlight that the detector considered in our work employs non-binary features (differently from all previous articles in the literature). Consequently, we think an interesting future work would be to compare the impact of the feature vector design (binary vs. non-binary) on the practical feasibility of adversarial malware creation.

We also point out that our specific implementation to address the inverse feature-mapping problem is not optimal as well in terms of plausibility. In fact, other work in the literature — such as the one by Pierazzi et al. [104] where portions of real code were injected through *automated software transplantation* — present, among others, the potential issue of injecting code that is out of context with respect to the original app. In our case, the main concern lies instead in the choice of injecting calls to class constructors. That is, we call new objects that are not used in the rest of the code. Therefore, a more plausible solution for future work could be the injection of method calls, which are more likely to be realistic even without being referenced next in the code.

Chapter 6

Conclusions

Machine learning for malware detection has proven to be an effective tool. However, its design process may hide pitfalls that undermine the deployment in real scenarios. Such concerns may refer to the usage of complex algorithms and feature sets that, all in all, allow neither interpreting what the models learn nor catching the actual characteristics of malware. Consequently, systems with these issues may learn spurious patterns and be particularly vulnerable to adversarial attacks.

In this scenario, the work of this thesis has put its main focus on the detection of Android ransomware, an impactful threat that had swift spread in the past few years. In this respect, I have first explored the detection performance of different sets of a single kind of feature, taking into account typical strategies from attackers to prevent detection, such as obfuscation. Then, using techniques from explainable machine learning, I have inspected the validity of the developed systems, with the ultimate goal of proposing effective methods to employ those techniques. In particular, experimental work has been done to start understanding how explainability can positively impact malware detection. Associated with the aim of improving the design process of machine learning-based systems, I have broadened my focus and co-authored work on adversarial attacks. In this case, I have collaborated in understanding potential relationships between explainable and adversarial learning, in order to assess models' vulnerability to evasion attacks. Moreover, by putting the hat of an attacker, the concrete feasibility of creating adversarial Android apps has been explored.

Overall, I claim the topics I have explored have turned out to be quite relevant and worth to be studied. The thesis has put major emphasis on experimental analysis. With this respect, I have realised the importance of beware of several different aspects within the set-up and design process of ML-based models. The work on the detection of Android ransomware has highlighted the necessity of examining the validity of the feature set from multiple perspectives, both the ML-specific and the domain-specific ones. In this sense, I point out that the proposed system API-based features present the typical disadvantages of static analysis. Moreover, a thorough assessment of their robustness against adversarial strategies has not been performed. Specifically, the practical feasibility of evasion attacks has only been tested against mimicry and random noise alterations. Therefore, future work should evaluate the vulnerability of system API calls against gradient-based evasion attacks. This way, it could also be possible to assess if this kind of strategy allows attackers to minimise the number of injected features; hence, preserving the plausibility of the adversarial

samples.
Differently from previous work in the literature, this thesis has the merit to consider practical attacks against non-binary features. However, the specific implementation to address the inverse feature-mapping problem is not refined enough in terms of plausibility compared to the newest works. Therefore, it would be interesting to inspect the effectiveness of more complex feature injections, such as *automated software transplantation*, against settings using non-binary features. Ultimately, such a further in-depth analysis would be relevant to understand if the architecture of operating systems or their components (e.g. the APIs) facilitate or prevent the practical accomplishment of evasion attacks.

The usage of explainable machine learning has revealed its potentialities in helping to interpret even security-specific elements of detectors. However, the considered two-class setting, where ransomware apps are evaluated against legitimate ones, is not able to fully capture behaviour *in the middle*. In other words, it would be relevant to also identify the characteristics that ransomware attacks do and do not share with generic malware. Moreover, the settings using system API *class* or *method* calls have not been inspected. Therefore, I point out that further new insights can be revealed by considering other setting variants. For example, it would be significant to check if classifiers using finer-grained API calls provide explanations that embed the same conceptual meaning of the coarser-grained settings.
Another limitation is related to the usage of gradient-based attribution techniques only, which do not allow understanding the interactions among the features. In this sense, combining different techniques may enable one to grasp the particular nuances of each kind of explanation. Even more, discovering high-level *concepts* might greatly improve the understanding of the applications under analysis.

Finally, the thesis has explored empirical ways to relate adversarial and explainable machine learning through proxy metrics. In this sense, I point out the possibility for future research work to first go deeper with the proposed experimental paths. Then, I also observe the necessity to develop more theoretically-sound frameworks that may embed the lessons learnt on the usage of explainable machine learning more systematically. Such an enhancement would ultimately allow generalising all the analyses and proposed methods of this thesis to other malware detection problems.

Bibliography

[1] ENISA. 'Analysis of the European R&D Priorities in Cybersecurity'. 2018. URL: https://www.enisa.europa.eu/publications/analysis-of-the-european-r-d-priorities-in-cybersecurity/at_download/fullReport.

[2] Europol. *Internet Organised Crime Threat Assessment (IOCTA) 2020*. Tech. rep. 2020. DOI: 10.22. URL: https://www.europol.europa.eu/activities-services/main-reports/internet-organised-crime-threat-assessment-iocta-2020.

[3] Thomas Haahr. *MEPs adopt Cybersecurity Act and want EU to counter IT threat from China*. 2019. URL: http://www.europarl.europa.eu/news/en/press-room/20190307IPR30694/meps-adopt-cybersecurity-act-and-want-eu-to-counter-it-threat-from-china.

[4] ENISA. *Threat Landscape 2020 - Research topics*. Tech. rep. 2020. URL: https://www.enisa.europa.eu/publications/enisa-threat-landscape-2020-research-topics.

[5] ENISA. *Threat Landscape Report 2015*. Tech. rep. 2016. URL: http://www.enisa.europa.eu/activities/risk-management/evolving-threat-environment/enisa-threat-landscape-mid-year-2013.

[6] ENISA. *Threat Landscape Report 2016*. Tech. rep. 2017. URL: https://www.enisa.europa.eu/publications/enisa-threat-landscape-report-2016.

[7] ENISA. *Threat Landscape Report 2017*. Tech. rep. 2018. URL: https://www.enisa.europa.eu/publications/enisa-threat-landscape-report-2017.

[8] ENISA. *Threat Landscape Report 2018*. Tech. rep. 2019. URL: https://www.enisa.europa.eu/publications/enisa-threat-landscape-report-2018.

[9] ENISA. *Threat Landscape - The year in review*. Tech. rep. 2020. URL: https://www.enisa.europa.eu/publications/year-in-review.

[10] ENISA. *Threat Landscape 2020 - Ransomware*. Tech. rep. 2020. URL: https://www.enisa.europa.eu/publications/ransomware.

[11] Michele Scalas and Giorgio Giacinto. 'On the Role of Explainable Machine Learning for Secure Smart Vehicles'. In: *2020 AEIT International Conference of Electrical and Electronic Technologies for Automotive (AEIT AUTOMOTIVE)*. Torino, Italy, Oct. 2020, pp. 1–6.

[12] Yousra Aafer, Wenliang Du and Heng Yin. 'DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android'. In: *Proc. of International Conference on Security and Privacy in Communication Networks ({SecureComm})*. 2013, pp. 86–103. DOI: 10.1007/978-3-319-04283-1{_}6. URL: http://link.springer.com/10.1007/978-3-319-04283-1_6.

[13] Martina Lindorfer, Matthias Neugschwandtner and Christian Platzer. 'Marvin: Efficient and Comprehensive Mobile App Classification Through Static and Dynamic Analysis'. In: *Proceedings of the 39th Annual International Computers, Software & Applications Conference (COMPSAC)*. 2015.

[14] Hao Peng, Chris Gates, Bhaskar Sarma, Ninghui Li, Yuan Qi, Rahul Potharaju, Cristina Nita-Rotaru and Ian Molloy. 'Using Probabilistic Generative Models for Ranking Risks of Android Apps'. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. 2012.

[15] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon and Konrad Rieck. 'Drebin: Effective and Explainable Detection of Android Malware in Your Pocket'. In: *Proceedings 2014 Network and Distributed System Security Symposium*. Reston, VA: Internet Society, 2014, pp. 23–26. ISBN: 1-891562-35-5. DOI: 10.14722/ndss.2014.23247. URL: https://www.ndss-symposium.org/ndss2014/programme/drebin-effective-and-explainable-detection-android-malware-your-pocket/.

[16] Sen Chen, Minhui Xue, Zhushou Tang, Lihua Xu and Haojin Zhu. 'StormDroid'. In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security - ASIA CCS '16*. ASIA CCS '16. New York, New York, USA: ACM Press, May 2016, pp. 377–388. ISBN: 9781450342339. DOI: 10.1145/2897845.2897860.

[17] Ambra Demontis, Marco Melis, Battista Biggio, Davide Maiorca, Daniel Arp, Konrad Rieck, Igino Corona, Giorgio Giacinto and Fabio Roli. 'Yes, Machine Learning Can Be More Secure! A Case Study on Android Malware Detection'. In: *IEEE Transactions on Dependable and Secure Computing* 16.4 (2017), pp. 1–1. ISSN: 1545-5971. DOI: 10.1109/TDSC.2017.2700270. URL: http://ieeexplore.ieee.org/document/7917369/.

[18] Michele Scalas, Davide Maiorca, Francesco Mercaldo, Corrado Aaron Visaggio, Fabio Martinelli and Giorgio Giacinto. 'On the effectiveness of system API-related information for Android ransomware detection'. In: *Computers & Security* 86 (Sept. 2019), pp. 168–182. ISSN: 01674048. DOI: 10.1016/j.cose.2019.06.004. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167404819301178%20http://arxiv.org/abs/1805.09563.

[19] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro and Konrad Rieck. 'Dos and Don'ts of Machine Learning in Computer Security'. In: (Oct. 2020). URL: http://arxiv.org/abs/2010.09470.

[20] Battista Biggio and Fabio Roli. 'Wild patterns: Ten years after the rise of adversarial machine learning'. In: *Pattern Recognition* 84 (2018), pp. 317–331. ISSN: 0031-3203. DOI: 10.1016/J.PATCOG.2018.07.023. URL: https://www.sciencedirect.com/science/article/pii/S0031320318302565.

[21] Nilesh Dalvi, Pedro Domingos, Mausam, Sumit Sanghai and Deepak Verma. 'Adversarial classification'. In: *Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. Seattle, 2004, pp. 99–108.

[22] Daniel Lowd and Christopher Meek. 'Adversarial Learning'. In: *Proc. 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. Chicago, IL, USA: ACM Press, 2005, pp. 641–647.

[23] Marco Barreno, Blaine Nelson, Russell Sears, Anthony D Joseph and J D Tygar. 'Can machine learning be secure?' In: *Proc. ACM Symp. Information, Computer and Comm. Sec.* ASIACCS '06. New York, NY, USA: ACM, 2006, pp. 16–25.

[24] Marco Barreno, Blaine Nelson, Anthony Joseph and J Tygar. 'The security of machine learning'. In: *Machine Learning* 81.2 (2010), pp. 121–148.

[25] Battista Biggio, Giorgio Fumera and Fabio Roli. 'Security Evaluation of Pattern Classifiers Under Attack'. In: *IEEE Transactions on Knowledge and Data Engineering* 26.4 (Apr. 2014), pp. 984–996. ISSN: 1041-4347.

[26] Battista Biggio, Blaine Nelson and Pavel Laskov. 'Poisoning attacks against support vector machines'. In: *29th Int{\textquoteright}l Conf. on M. Learning (ICML)*. Ed. by John Langford and Joelle Pineau. Omnipress. Omnipress, 2012.

[27] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto and Fabio Roli. 'Evasion Attacks against Machine Learning at Test Time'. In: *Joint European conference on machine learning and knowledge discovery in databases*. Vol. 8190. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 387–402. DOI: 10.1007/978-3-642-40994-3{_}25. URL: http://link.springer.com/10.1007/978-3-642-40994-3_25.

[28] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow and Rob Fergus. 'Intriguing properties of neural networks'. In: *International Conference on Learning Representations*. 2014. URL: http://arxiv.org/abs/1312.6199.

[29] Ian J Goodfellow, Jonathon Shlens and Christian Szegedy. 'Explaining and Harnessing Adversarial Examples'. In: *International Conference on Learning Representations*. 2015.

[30] Davide Maiorca, Battista Biggio and Giorgio Giacinto. 'Towards adversarial malware detection: Lessons learned from PDF-based attacks'. In: *ACM Computing Surveys (CSUR)* 52.4 (2019), pp. 1–36.

[31] Nedim Šrndic and Pavel Laskov. 'Practical Evasion of a Learning-Based Classifier: A Case Study'. In: *Proc. 2014 IEEE Symp. Security and Privacy*. SP '14. Washington, DC, USA: IEEE CS, 2014, pp. 197–211.

[32] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes and Patrick McDaniel. 'Adversarial Examples for Malware Detection'. In: *European Symposium on Research in Computer Security (ESORICS)*. Vol. 10493 LNCS. Springer Verlag, 2017, pp. 62–79. ISBN: 9783319663982. DOI: 10.1007/978-3-319-66399-9{_}4. URL: http://link.springer.com/10.1007/978-3-319-66399-9_4.

[33] Amina Adadi and Mohammed Berrada. 'Peeking Inside the Black-Box: A Survey on Explainable Artificial Intelligence (XAI)'. In: *IEEE Access* 6 (2018), pp. 52138–52160. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2870052.

[34] Alejandro Barredo Arrieta, Natalia Díaz-Rodríguez, Javier Del Ser, Adrien Bennetot, Siham Tabik, Alberto Barbado, Salvador Garcia, Sergio Gil-Lopez, Daniel Molina, Richard Benjamins, Raja Chatila and Francisco Herrera. 'Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI'. In: *Information Fusion* 58.October 2019 (June 2020), pp. 82–115. ISSN: 15662535. DOI: 10.1016/j.inffus.2019.12.012.

[35] Finale Doshi-Velez and Been Kim. 'Towards A Rigorous Science of Interpretable Machine Learning'. In: *arXiv preprint arXiv:1702.08608* (2017).

[36] Alun Preece, Dan Harborne, Dave Braines, Richard Tomsett and Supriyo Chakraborty. 'Stakeholders in Explainable AI'. In: *AAAI Fall Symposium on Artificial Intelligence in Government and Public Sector*. Arlington, Virginia, USA, 2018.

[37] Michele Scalas and Giorgio Giacinto. 'Automotive Cybersecurity: Foundations for Next-Generation Vehicles'. In: *2019 2nd International Conference on new Trends in Computing Sciences (ICTCS)*. IEEE, Oct. 2019, pp. 1–6. ISBN: 978-1-7281-2882-5. DOI: 10.1109/ICTCS.2019.8923077. URL: https://ieeexplore.ieee.org/document/8923077/.

[38] Cynthia Rudin. 'Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead'. In: *Nature Machine Intelligence* 1.5 (May 2019), pp. 206–215. ISSN: 2522-5839. DOI: 10.1038/s42256-019-0048-x. URL: http://www.nature.com/articles/s42256-019-0048-x.

[39] Jon Arne Glomsrud, André Ødegårdstuen, Asun Lera St. Clair and OYvind Smogeli. 'Trustworthy versus Explainable AI in Autonomous Vessels'. In: *Proceedings of the International Seminar on Safety and Security of Autonomous Vessels (ISSAV) and European STAMP Workshop and Conference (ESWC) 2019*. September. Sciendo, Dec. 2020, pp. 37–47. DOI: 10.2478/9788395669606-004.

[40] Been Kim, Martin Wattenberg, Justin Gilmer, Carrie Cai, James Wexler, Fernanda Viegas and Rory Sayres. 'Interpretability Beyond Feature Attribution: Quantitative Testing with Concept Activation Vectors (TCAV)'. In: *35th International Conference on Machine Learning (ICML 2018)*. Vol. 80. Stockholm, July 2018, pp. 2668–2677. ISBN: 9781510867963.

[41] Amirata Ghorbani, James Wexler, James Zou and Been Kim. 'Towards Automatic Concept-based Explanations'. In: *Advances in Neural Information Processing Systems*. Vancouver: Curran Associates, Inc., Feb. 2019, pp. 9273–9282.

[42] Marco Tulio Ribeiro, Sameer Singh and Carlos Guestrin. '"Why Should I Trust You?"' In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '16*. KDD '16. New York, NY, USA: ACM Press, Aug. 2016, pp. 1135–1144. ISBN: 9781450342322. DOI: 10.1145/2939672.2939778. URL: http://dl.acm.org/citation.cfm?doid=2939672.2939778.

[43] Scott M. Lundberg, Su-In Lee, Paul G Allen and Su-In Lee. 'A Unified Approach to Interpreting Model Predictions'. In: *Advances in Neural Information Processing Systems 30*. Curran Associates, Inc., 2017, pp. 4765–4774.

[44] Pang Wei Koh and Percy Liang. 'Understanding Black-box Predictions via Influence Functions'. In: *Proceedings of the 34th International Conference on Machine Learning - Volume 70* (Mar. 2017), pp. 1885–1894.

[45] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang and Xinyu Xing. 'LEMNA'. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: ACM, Jan. 2018, pp. 364–379. ISBN: 9781450356930. DOI: 10.1145/3243734.3243792. URL: http://dl.acm.org/citation.cfm?doid=3243734.3243792%20https://dl.acm.org/doi/10.1145/3243734.3243792.

[46] David Baehrens, Timon Schroeter, Stefan Harmeling, Motoaki Kawanabe, Katja Hansen and Klaus-Robert Müller. 'How to Explain Individual Classification Decisions'. In: *J. Mach. Learn. Res.* 11 (2010), pp. 1803–1831.

[47] Avanti Shrikumar, Peyton Greenside, Anna Shcherbina and Anshul Kundaje. *Not Just a Black Box: Learning Important Features Through Propagating Activation Differences*. 2016.

[48] Mukund Sundararajan, Ankur Taly and Qiqi Yan. 'Axiomatic Attribution for Deep Networks'. In: *Proceedings of the 34th International Conference on Machine Learning*. Sidney: JMLR.org, Mar. 2017, pp. 3319–3328. URL: http://arxiv.org/abs/1703.01365.

[49] Kaspersky. *Mobile Malware Evolution 2019*. 2020. URL: https://securelist.com/mobile-malware-evolution-2019/96280/.

[50] Trend Micro. *The 2019 Mobile Threat Landscape - Security Roundup*. URL: https://www.trendmicro.com/vinfo/us/security/research-and-analysis/threat-reports/roundup/review-refocus-and-recalibrate-the-2019-mobile-threat-landscape.

[51] Kaspersky. *IT threat evolution Q2 2020. Mobile statistics*. 2020. URL: https://securelist.com/it-threat-evolution-q2-2020-mobile-statistics/98337/.

[52] Kaspersky. 'Mobile Malware Evolution 2016'. 2017. URL: https://securelist.com/mobile-malware-evolution-2016/77681/.

[53] Kaspersky. *Mobile Malware Evolution 2017*. 2018. URL: https://securelist.com/mobile-malware-review-2017/84139/.

[54] Kaspersky. *Mobile Malware Evolution 2018*. 2019. URL: https://securelist.com/mobile-malware-evolution-2018/89689/.

[55] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh and Lorenzo Cavallaro. 'The Evolution of Android Malware and Android Analysis Techniques'. In: *ACM Computing Surveys* 49.4 (Feb. 2017), pp. 1–41. ISSN: 0360-0300. DOI: 10.1145/3017427.

[56] Mohammed Talal, A. A. Zaidan, B. B. Zaidan, O. S. Albahri, M. A. Alsalem, A. S. Albahri, A. H. Alamoodi, M. L. M. Kiah, F. M. Jumaah and Mussab Alaa. 'Comprehensive review and analysis of anti-malware apps for smartphones'. In: *Telecommunication Systems* 72.2 (Oct. 2019), pp. 285–337. ISSN: 1018-4864. DOI: 10.1007/s11235-019-00575-7. URL: http://link.springer.com/10.1007/s11235-019-00575-7.

[57] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau and Patrick McDaniel. '{FlowDroid}: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for {Android} apps'. In: *Proceedings of the 35th {ACM} {SIGPLAN} {Conference} on {Programming} {Language} {Design} and {Implementation} - {PLDI} '14*. ACM Press, 2013, pp. 259–269. ISBN: 978-1-4503-2784-8. DOI: 10.1145/2594291.2594299. URL: http://dl.acm.org/citation.cfm?doid=2594291.2594299.

[58] Yu Feng, Saswat Anand, Isil Dillig and Alex Aiken. 'Apposcopy: semantics-based detection of {Android} malware through static analysis'. In: *Proceedings of the 22nd {ACM} {SIGSOFT} {International} {Symposium} on {Foundations} of {Software} {Engineering} - {FSE} 2014*. ACM Press, 2014, pp. 576–587. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635869. URL: http://dl.acm.org/citation.cfm?doid=2635868.2635869.

[59] Joshua Garcia, Mahmoud Hammad and Sam Malek. 'Lightweight, Obfuscation-Resilient Detection and Family Identification of Android Malware'. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 26.3 (2018), 11:1–11:29. ISSN: 1049-331X. DOI: 10.1145/3162625.

[60] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X Sean Wang and Binyu Zang. 'Vetting undesirable behaviors in android apps with permission use analysis'. In: *Proceedings of the 2013 {ACM} {SIGSAC} conference on {Computer} & communications security - {CCS} '13*. ACM Press, 2013, pp. 611–622. ISBN: 978-1-4503-2477-9. DOI: 10.1145/2508859.2516689. URL: http://dl.acm.org/citation.cfm?doid=2508859.2516689.

[61] Kimberly Tam, Salahuddin J Khan, Aristide Fattori and Lorenzo Cavallaro. 'CopperDroid: Automatic Reconstruction of Android Malware Behaviors'. In: *Proc. 22nd Annual Network & Distributed System Security Symposium ({NDSS})*. The Internet Society, 2015.

[62] Jing Chen, Chiheng Wang, Ziming Zhao, Kai Chen, Ruiying Du and Gail-Joon Ahn. 'Uncovering the Face of Android Ransomware: Characterization and Real-Time Detection'. In: *IEEE Transactions on Information Forensics and Security (TIFS)* 13.5 (2018), pp. 1286–1300. ISSN: 1556-6013. DOI: 10.1109/TIFS.2017.2787905.

[63] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie and William Enck. 'AppContext: Differentiating Malicious and Benign Mobile App Behaviors Using Context'. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. ICSE '15. Piscataway, NJ, USA: IEEE, May 2015, pp. 303–313. ISBN: 978-1-4799-1934-5. DOI: 10.1109/ICSE.2015.50.

[64] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer and Eric Bodden. 'Mining Apps for Abnormal Usage of Sensitive Data'. In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 426–436. ISBN: 978-1-4799-1934-5.

[65] Hao Zhang, Danfeng Daphne Yao and Naren Ramakrishnan. 'Detection of Stealthy Malware Activities with Traffic Causality and Scalable Triggering Relation Discovery'. In: *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*. ASIA CCS '14. New York, NY, USA: ACM, 2014, pp. 39–50. ISBN: 978-1-4503-2800-5. DOI: 10.1145/2590296.2590309.

[66] Hao Zhang, Danfeng (Daphne) Yao and Naren Ramakrishnan. 'Causality-based Sensemaking of Network Traffic for Android Application Security'. In: *Proceedings of the 2016 ACM Workshop on Artificial Intelligence and Security*. AISec '16. New York, NY, USA: ACM, 2016, pp. 47–58. ISBN: 978-1-4503-4573-6. DOI: 10.1145/2996758.2996760.

[67] Kevin Sun. *Google Play Apps Drop Anubis, Use Motion-based Evasion*. URL: https://www.trendmicro.com/en_us/research/19/a/google-play-apps-drop-anubis-banking-malware-use-motion-based-evasion-tactics.html?_ga=2.125935757.1289840044.1606563087-979093796.1605611851.

[68] Michele Scalas, Konrad Rieck and Giorgio Giacinto. 'Explanation-driven Characterisation of Android Ransomware'. In: *ICPR'2020 Workshop on Explainable Deep Learning - AI*. 2020.

[69] Nicoló Andronio, Stefano Zanero and Federico Maggi. 'HelDroid: Dissecting and Detecting Mobile Ransomware'. In: *Recent Advances in Intrusion Detection (RAID)*. Vol. 9404. Springer, 2015, pp. 382–404. ISBN: 9783319263618. DOI: 10.1007/978-3-319-26362-5{_}18. URL: http://link.springer.com/10.1007/978-3-319-26362-5_18.

[70] Chengyu Zheng, Nicola Dellarocca, Niccolò Andronio, Stefano Zanero and Federico Maggi. 'GreatEatlon: Fast, Static Detection of Mobile Ransomware'. In: *SecureComm*. Vol. 198. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Springer, Oct. 2017, pp. 617–636. ISBN: 9783319596075. DOI: 10.1007/978-3-319-59608-2{_}34.

[71] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu and Giorgio Giacinto. 'Stealth Attacks: An Extended Insight into the Obfuscation Effects on Android Malware'. In: *Computers & Security* 51.C (2015), pp. 16–31. ISSN: 0167-4048. DOI: 10.1016/j.cose.2015.02.007.

[72] Aniello Cimitile, Francesco Mercaldo, Vittoria Nardone, Antonella Santone and Corrado Aaron Visaggio. 'Talos: no more ransomware victims with formal methods'. In: *International Journal of Information Security* 17.6 (Nov. 2018), pp. 719–738. ISSN: 1615-5262. DOI: 10.1007/s10207-017-0398-5.

[73] Amirhossein Gharib and Ali Ghorbani. 'DNA-Droid: A Real-Time Android Ransomware Detection Framework'. In: *International Conference on Network and System Security*. Springer, Cham, 2017, pp. 184–198. DOI: 10.1007/978-3-319-64701-2{_}14.

[74] Sanggeun Song, Bongjoon Kim and Sangjun Lee. 'The Effective Ransomware Prevention Technique Using Process Monitoring on Android Platform'. In: *Mobile Information Systems* 2016 (May 2016), pp. 1–9. ISSN: 1574-017X. DOI: 10.1155/2016/2946735. URL: http://www.hindawi.com/journals/misy/2016/2946735/.

[75] Tianda Yang, Yu Yang, Kai Qian, Dan Chia-Tien Lo, Ying Qian and Lixin Tao. 'Automated Detection and Analysis for Android Ransomware'. In: *2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS)*. IEEE. 2015, pp. 1338–1343.

[76] Davide Maiorca, Francesco Mercaldo, Giorgio Giacinto, Corrado Aaron Visaggio and Fabio Martinelli. 'R-PackDroid: API Package-Based Characterization and Detection of Mobile Ransomware'. In: *Proceedings of the Symposium on Applied Computing - SAC '17*. SAC '17. New York, New York, USA: ACM Press, 2017, pp. 1718–1723. ISBN: 9781450344869. DOI: 10.1145/3019612.3019793. URL: http://dl.acm.org/citation.cfm?doid=3019612.3019793.

[77] Marcos Sebastián, Richard Rivera, Platon Kotzias and Juan Caballero. 'AVclass: A Tool for Massive Malware Labeling'. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 9854 LNCS. Springer Verlag, 2016, pp. 230–253. ISBN: 9783319457185. DOI: 10.1007/978-3-319-45719-2{_}11.

[78] Yajin Zhou and Xuxian Jiang. 'Dissecting android malware: Characterization and evolution'. In: *IEEE Symposium on Security and Privacy*. IEEE. 2012, pp. 95–109.

[79] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein and Yves Le Traon. 'AndroZoo: Collecting Millions of Android Apps for the Research Community'. In: *Proceedings of the 13th International Conference on Mining Software Repositories*. MSR '16. New York, NY, USA: ACM, 2016, pp. 468–471. ISBN: 978-1-4503-4186-8. DOI: 10.1145/2901739.2903508.

[80] Mansour Ahmadi, Angelo Sotgiu and Giorgio Giacinto. 'IntelliAV: Toward the Feasibility of Building Intelligent Anti-malware on Android Devices'. In: *Machine Learning and Knowledge Extraction*. Ed. by Andreas Holzinger, Peter Kieseberg, A Min Tjoa and Edgar Weippl. Vol. 86. Cham: Springer International Publishing, Sept. 2017, pp. 137–154. ISBN: 978-3-319-66808-6. DOI: 10.1007/978-3-319-66808-6{_}10.

[81] Marco Melis, Davide Maiorca, Battista Biggio, Giorgio Giacinto and Fabio Roli. 'Explaining Black-box Android Malware Detection'. In: *2018 26th European Signal Processing Conference (EUSIPCO)*. IEEE, Sept. 2018, pp. 524–528. ISBN: 978-9-0827-9701-5. DOI: 10.23919/EUSIPCO.2018.8553598. URL: https://ieeexplore.ieee.org/document/8553598/.

[82] Battista Biggio, Giorgio Fumera and Fabio Roli. 'Pattern Recognition Systems under Attack: Design Issues and Research Challenges'. In: *IJPRAI* 28.7 (2014).

[83] Alexander Warnecke, Daniel Arp, Christian Wressnegger and Konrad Rieck. 'Evaluating Explanation Methods for Deep Learning in Security'. In: *5th IEEE European Symposium on Security and Privacy (Euro S&P 2020)*. Genova, Sept. 2020.

[84] Julius Adebayo, Justin Gilmer, Michael Muelly, Ian Goodfellow, Moritz Hardt and Been Kim. 'Sanity Checks for Saliency Maps'. In: *Advances in Neural Information Processing Systems 31*. Curran Associates, Inc., Oct. 2018, pp. 9505–9515.

[85] Isaac Lage, Emily Chen, Jeffrey He, Menaka Narayanan, Been Kim, Samuel J Gershman and Finale Doshi-Velez. 'An Evaluation of the Human-Interpretability of Explanation'. In: *Proceedings of the AAAI Conference on Human Computation and Crowdsourcing*. Vol. 7. Honolulu: AAAI Press, Jan. 2019, pp. 59–67.

[86] *Android API Reference*. URL: https://developer.android.com/reference/packages.

[87] Marco Melis, Ambra Demontis, Maura Pintor, Angelo Sotgiu and Battista Biggio. 'secml: A Python Library for Secure and Explainable Machine Learning'. In: *arXiv preprint arXiv:1912.10013* (Dec. 2019). URL: http://arxiv.org/abs/1912.10013.

[88] Marco Ancona. *DeepExplain*. URL: https://github.com/marcoancona/DeepExplain.

[89] Maximilian Alber, Sebastian Lapuschkin, Philipp Seegerer, Miriam Hägele, Kristof T. Schütt, Grégoire Montavon, Wojciech Samek, Klaus-Robert Müller, Sven Dähne and Pieter-Jan Kindermans. *iNNvestigate neural networks!* Aug. 2018. URL: http://arxiv.org/abs/1808.04260.

[90] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder and Lorenzo Cavallaro. 'TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time'. In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA, USA: USENIX Association, Aug. 2019, pp. 729–746.

[91] Alejandro Calleja, Alejandro Martin, Hector D Menendez, Juan Tapiador and David Clark. 'Picking on the family: Disrupting android malware triage by forcing misclassification'. In: *Expert Systems with Applications* 95 (2018), pp. 113–126.

[92] Marco Melis, Michele Scalas, Ambra Demontis, Davide Maiorca, Battista Biggio, Giorgio Giacinto and Fabio Roli. 'Do Gradient-based Explanations Tell Anything About Adversarial Robustness to Android Malware?' In: *arXiv preprint arXiv:2005.01452* (May 2020). URL: http://arxiv.org/abs/2005.01452.

[93] Fabrizio Cara, Michele Scalas, Giorgio Giacinto and Davide Maiorca. 'On the Feasibility of Adversarial Sample Creation Using the Android System API'. In: *Information 2020, Vol. 11, Page 433* 11.9 (Sept. 2020), p. 433. DOI: 10.3390/info11090433. URL: www.mdpi.com/journal/information.

[94] Aleksander Kolcz and Choon Hui Teo. 'Feature Weighting for Improved Classifier Robustness'. In: *Sixth Conference on Email and Anti-Spam (CEAS)*. Mountain View, CA, USA, 2009.

[95] Battista Biggio, Giorgio Fumera and Fabio Roli. 'Multiple Classifier Systems for Robust Classifier Design in Adversarial Environments'. In: *Int'l J. Mach. Learn. and Cybernetics* 1.1 (2010), pp. 27–41.

[96] Ambra Demontis, Paolo Russu, Battista Biggio, Giorgio Fumera and Fabio Roli. 'On Security and Sparsity of Linear Classifiers for Adversarial Settings'. In: *Joint IAPR Int'l Workshop on Structural, Syntactic, and Statistical Pattern Recognition*. Ed. by Antonio Robles-Kelly, Marco Loog, Battista Biggio, Francisco Escolano and Richard Wilson. Vol. 10029. LNCS. Cham: Springer International Publishing, 2016, pp. 322–332.

[97] Ambra Demontis, Marco Melis, Maura Pintor, Matthew Jagielski, Battista Biggio, Alina Oprea, Cristina Nita-Rotaru and Fabio Roli. 'Why Do Adversarial Attacks Transfer? Explaining Transferability of Evasion and Poisoning Attacks'. In: *28th {USENIX} Security Symposium ({USENIX} Security 19)*. Santa Clara, CA, USA: {USENIX} Association, Aug. 2019, pp. 321–338. ISBN: 978-1-939133-06-9.

[98] Jiefeng Chen, Xi Wu, Vaibhav Rastogi, Yingyu Liang and Somesh Jha. 'Robust Attribution Regularization'. In: *Advances in Neural Information Processing Systems*. 2019, pp. 14300–14310.

[99] Martina Lindorfer, Matthias Neugschwandtner and Christian Platzer. 'MARVIN: Efficient and Comprehensive Mobile App Classification through Static and Dynamic Analysis'. In: *2015 IEEE 39th Annual Computer Software and Applications Conference*. Vol. 2. IEEE, July 2015, pp. 422–433. ISBN: 978-1-4673-6564-2. DOI: 10.1109/COMPSAC.2015.103. URL: http://ieeexplore.ieee.org/document/7273650/.

[100] Haipeng Cai, Na Meng, Barbara Ryder and Daphne Yao. 'Droidcat: Effective android malware detection and categorization via app-level profiling'. In: *IEEE Transactions on Information Forensics and Security* 14.6 (2018), pp. 1455–1470.

[101] Marco Melis, Ambra Demontis, Battista Biggio, Gavin Brown, Giorgio Fumera and Fabio Roli. 'Is Deep Learning Safe for Robot Vision? Adversarial Examples against the iCub Humanoid'. In: *ICCV Workshop on Vision in Practice on Autonomous Robots ({ViPAR})*. 2017.

[102] Davide Maiorca, Igino Corona and Giorgio Giacinto. 'Looking at the bag is not enough to find the bomb: An evasion of structural methods for malicious PDF files detection'. In: *ASIA CCS 2013 - Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*. New York, New York, USA: ACM Press, 2013, pp. 119–129. ISBN: 9781450317672. DOI: 10.1145/2484313.2484327.

[103] Erwin Quiring, Alwin Maier and Konrad Rieck. 'Misleading Authorship Attribution of Source Code using Adversarial Learning | USENIX'. In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 479–496. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/quiring.

[104] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi and Lorenzo Cavallaro. 'Intriguing Properties of Adversarial ML Attacks in the Problem Space'. In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. IEEE, May 2020, pp. 1332–1349. ISBN: 978-1-7281-3497-0. DOI: 10.1109/SP40000.2020.00073. URL: https://ieeexplore.ieee.org/document/9152781/.

[105] Wei Song, Xuezixiang Li, Sadia Afroz, Deepali Garg, Dmitry Kuznetsov and Heng Yin. 'Automatic Generation of Adversarial Examples for Interpreting Malware Classifiers'. In: (Mar. 2020). URL: http://arxiv.org/abs/2003.03100.

[106] *Smali*. URL: https://github.com/JesusFreke/smali.

[107] *ApkTool*. URL: https://ibotpeaches.github.io/Apktool.

[108] Mila Dalla Preda, Matias Madou, Koen De Bosschere and Roberto Giacobazzi. 'Opaque Predicates Detection by Abstract Interpretation'. In: *Algebraic {Methodology} and {Software} {Technology}*. Vol. 4019. Springer Berlin Heidelberg, 2006, pp. 81–95. DOI: 10.1007/11784180{_}9. URL: http://link.springer.com/10.1007/11784180_9.

[109] Jiang Ming, Dongpeng Xu, Li Wang and Dinghao Wu. 'LOOP: Logic-Oriented Opaque Predicate Detection in Obfuscated Binary Code'. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*. New York, New York, USA: ACM Press, 2015, pp. 757–768. ISBN: 9781450338325. DOI: 10.1145/2810103.2813617. URL: http://dl.acm.org/citation.cfm?doid=2810103.2813617.

[110] Wei Yang, Deguang Kong, Tao Xie and Carl A. Gunter. 'Malware Detection in Adversarial Settings'. In: *Proceedings of the 33rd Annual Computer Security Applications Conference*. Vol. Part F1325. New York, NY, USA: ACM, Dec. 2017, pp. 288–302. ISBN: 9781450353458. DOI: 10.1145/3134600.3134642. URL: https://dl.acm.org/doi/10.1145/3134600.3134642.

[111] Inigo Incer, Michael Theodorides, Sadia Afroz and David Wagner. 'Adversarially Robust Malware Detection Using Monotonic Classification'. In: *Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics*. Vol. 10. New York, NY, USA: ACM, 2018. ISBN: 9781450356343. URL: https://doi.org/10.1145/3180445.3180449.

[112] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha and Ananthram Swami. 'Distillation as a Defense to Adversarial Perturbations Against Deep Neural Networks'. In: *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2016, pp. 582–597. ISBN: 978-1-5090-0824-7. DOI: 10.1109/SP.2016.41. URL: http://ieeexplore.ieee.org/document/7546524/.

[113] Ishai Rosenberg, Asaf Shabtai, Lior Rokach and Yuval Elovici. 'Generic black-box end-to-end attack against state of the art API call based malware classifiers'. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 11050 LNCS. Springer Verlag, Sept. 2018, pp. 490–510. ISBN: 9783030004699. DOI: 10.1007/978-3-030-00470-5{_}23.

[114] Weiwei Hu and Ying Tan. 'Generating {Adversarial} {Malware} {Examples} for {Black}-{Box} {Attacks} {Based} on {GAN}'. In: *arXiv:1702.05983 [cs]* (Feb. 2017). URL: http://arxiv.org/abs/1702.05983.

[115] Jerry Li, Aleksander Madry, John Peebles and Ludwig Schmidt. 'On the Limitations of First-Order Approximation in GAN Dynamics'. In: *arXiv preprint arXiv:1706.09884* (June 2017), p. 9. URL: http://arxiv.org/abs/1706.09884.