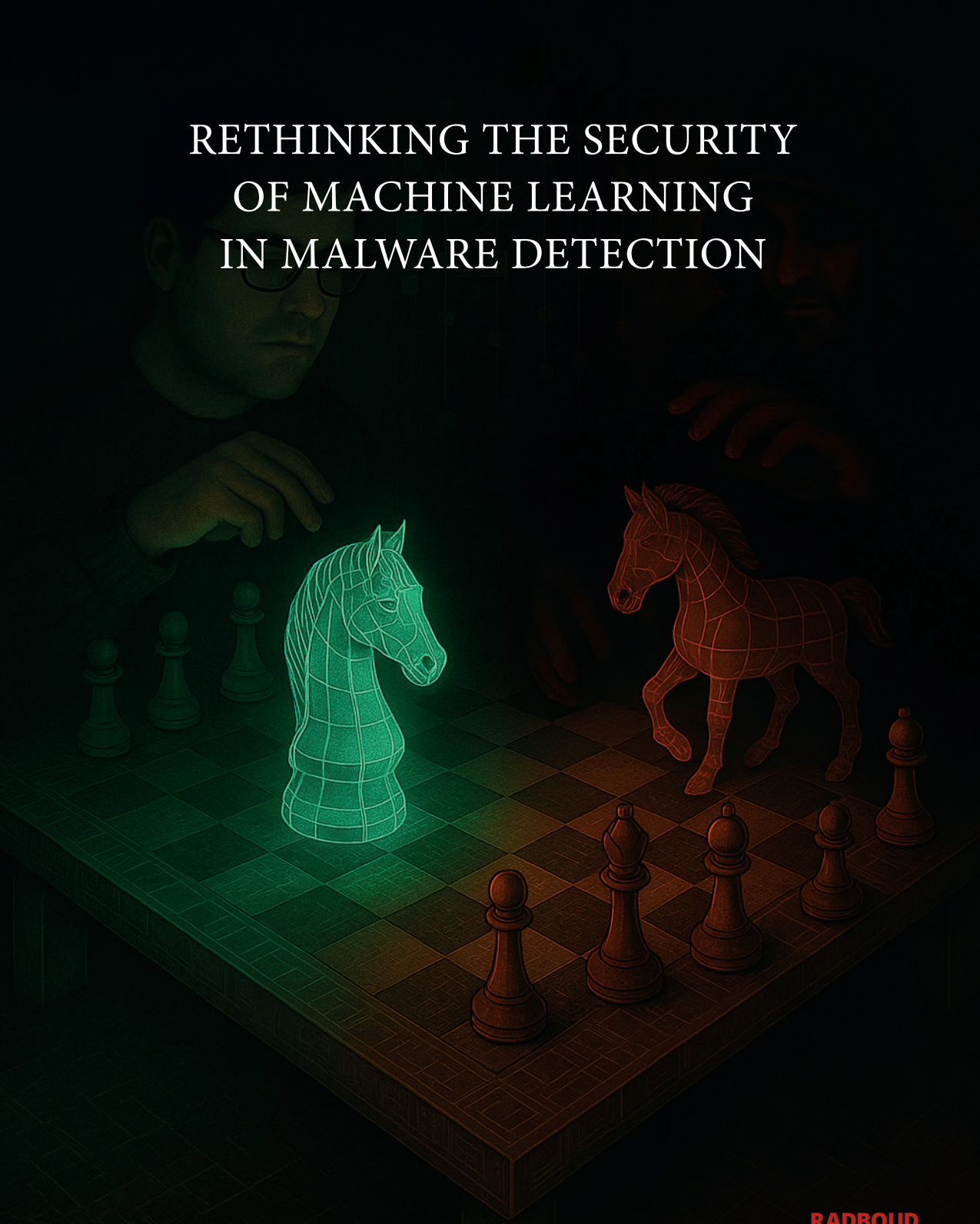
RETHINKING THE SECURITY OF MACHINE LEARNING IN MALWARE DETECTION



Hamid Bostani

Institute for Computing and Information Sciences

RADBOUD UNIVERSITY PRESS

Radboud Dissertation Series

Rethinking the Security of Machine Learning in Malware Detection

Hamid Bostani

This dissertation presents research conducted within the Digital Security research group at the Institute for Computing and Information Sciences, Radboud University.

RETHINKING THE SECURITY OF MACHINE LEARNING IN MALWARE DETECTION
Hamid Bostani

Radboud Dissertation Series
ISSN: 2950-2772 (Online); 2950-2780 (Print)

Published by RADBOUD UNIVERSITY PRESS
Postbus 9100, 6500 HA Nijmegen, The Netherlands
www.radbouduniversitypress.nl

Design: Hamid Bostani
Cover: Hamid Bostani
Printing: DPN Rikken/Pumbo

ISBN: 9789465151304
DOI: 10.54195/9789465151304
Free download at: https://doi.org/10.54195/9789465151304

RADBOUD
UNIVERSITY
PRESS

Rethinking the Security of Machine Learning in Malware Detection

Proefschrift

ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus prof. dr. J.M. Sanders,
volgens besluit van het college voor promoties
in het openbaar te verdedigen op

dinsdag 9 september 2025
om 14:30 uur precies

door

Hamid Bostani

geboren op 29 november 1984
te Shiraz, Iran

Promotoren:

Dr. ir. Erik Poll
Prof. dr. Veelasha Moonsamy (Ruhr-Universität Bochum, Duitsland)

Manuscriptcommissie:

Prof. dr. Lejla Batina (voorzitter)
Prof. dr. Konrad Rieck (Technische Universität Berlin, Duitsland)
Prof. dr. Mauro Conti (Università degli studi di Padova, Italië)
Dr. Juan Caballero (Instituto IMDEA Materiales, Spanje)
Dr. Jason Xue (CSIRO, Data61, Australië)

Rethinking the Security of Machine Learning in Malware Detection

Dissertation

to obtain the degree of doctor
from Radboud University Nijmegen
on the authority of the Rector Magnificus prof. dr. J.M. Sanders,
according to the decision of the Doctorate Board
to be defended in public on

Tuesday, September 9, 2025
at 2:30 pm

by

Hamid Bostani

born on November 29, 1984
in Shiraz, Iran

Supervisors:

Dr. ir. Erik Poll
Prof. dr. Veelasha Moonsamy (Ruhr University Bochum, Germany)

Manuscript Committee:

Prof. dr. Lejla Batina (Chair)
Prof. dr. Konrad Rieck (Technical University of Berlin, Germany)
Prof. dr. Mauro Conti (University of Padua, Italy)
Dr. Juan Caballero (IMDEA Software Institute, Spain)
Dr. Jason Xue (CSIRO, Data61, Australia)

Knowledge is a great treasure that does not diminish with spending.

Imam Ali (peace be upon him)

Contents

Summary

*Mal*icious soft*ware* (malware) continues to pose a growing and evolving threat to systems security, exploiting vulnerabilities to cause significant harm. In response, Machine Learning (ML) has emerged as a promising tool in malware detection, being leveraged in both static and dynamic analysis to identify malicious programs. However, despite the effectiveness of ML in improving malware detection, it faces inherent challenges, particularly adversarial vulnerabilities that evasion attacks exploit. These adversarial attacks involve crafting *adversarial malware* by manipulating malware to bypass detection systems while retaining its malicious functionality. As a result, evasion attacks undermine the robustness of ML classifiers, necessitating the development of defenses that can withstand such sophisticated threats.

This dissertation rethinks the security of ML-based malware detection systems against evasion attacks by incorporating realistic perspectives from both attackers and defenders. Specifically, this work highlights the challenges of integrating realism into both attack and defense strategies, such as identifying feasible adversarial vulnerabilities in malware classifiers or fairly evaluating the robustness of malware detection. By reassessing these aspects, we aim to offer a more practical viewpoint for strengthening ML-based malware detection against realistic evasion attacks. The research begins by exploring realistic threat models of evasion attacks, pushing the boundaries of current understanding and testing of such adversarial attacks. A key contribution on the offensive side is the proposal of a practical evasion attack, named EvadeDroid, that tackles the fundamental challenge of creating evasion techniques independent of the target model. This independence is crucial, as it allows the attacks to be applicable in real-world settings, where malware detectors typically operate as black boxes, returning only hard labels. The dissertation highlights that adversarial malware must not only preserve its malicious functionality but also ensure that the manipulation remains undetectable during preprocessing, enabling it to bypass detection. Additionally, EvadeDroid is optimized for efficiency, minimizing query costs and appearing plausible by injecting adversarial payloads prepared based on realistic, existing code, making it more practical and harder to defend against.

On the defensive side, the dissertation investigates methods to efficiently and effectively identify the vulnerabilities of malware classifiers against realistic evasion attacks and explores reliable defenses to protect malware detection systems from such adversarial attacks. A significant contribution of this part of the research is identifying feasible vulnerabilities in malware classifiers by focusing on generating potential adversarial malware directly in the feature space, rather than the problem space, thus providing more efficient and broader protection. The dissertation also investigates the issue of spurious correlations—misleading patterns that malware classifiers may mistakenly learn. By ensuring that the features used in detection better align with the true functionality of malware, classifiers can be improved to resist adversarial manipulation and generalize more effectively. Finally, the research delves into adversarial training (AT), a common technique for strengthening malware classifiers

against adversarial attacks. This investigation identifies key dimensions for exploring and evaluating AT, such as feature representations and the realism of adversarial examples, by proposing a unified framework that helps identify essential factors influencing the success of AT, such as model flexibility. The dissertation underscores the importance of considering all these factors in conjunction, rather than in isolation, to achieve meaningful improvements in classifier robustness.

In conclusion, the thesis offers a practical and forward-thinking approach to defending against adversarial threats in malware detection. It emphasizes the need for more realistic threat models and reliable defense strategies. By addressing the complexities of practical evasion attacks and providing a comprehensive roadmap for building resilient systems, this work contributes to the development of more secure and effective ML-based malware detection systems that can withstand the ever-evolving landscape of cyber threats. The thesis concludes by suggesting several potential research directions, particularly focusing on how the quality of training data influences the development of more robust malware classifiers.

Samenvatting

*Mal*icious soft*ware* (malware) blijft een groeiende en zich ontwikkelende bedreiging vormen voor de systeembeveiliging, waarbij kwetsbaarheden worden misbruikt om aanzienlijke schade aan te richten. In reactie hierop is Machine Learning (ML) naar voren gekomen als een veelbelovend hulpmiddel voor malwaredetectie, waarbij het wordt ingezet in zowel statische als dynamische analyses om kwaadaardige programma's te identificeren. Hoewel ML effectief is gebleken in het verbeteren van malwaredetectie, kampt het met inherente uitdagingen, met name kwetsbaarheden voor adversariële aanvallen die gebruikmaken van ontwijkingstechnieken. Deze aanvallen bestaan uit het vervaardigen van *adversariële malware*, waarbij malware zodanig wordt gemanipuleerd dat detectiesystemen worden omzeild, terwijl de kwaadaardige functionaliteit behouden blijft. Hierdoor ondermijnen ontwijkingsaanvallen de robuustheid van ML-classificatiemodellen, wat de ontwikkeling vereist van verdedigingsmechanismen die bestand zijn tegen zulke geavanceerde bedreigingen.

Dit proefschrift heroverweegt de beveiliging van op ML-gebaseerde malwaredetectiesystemen tegen ontwijkingsaanvallen, door realistische perspectieven van zowel aanvallers als verdedigers te integreren. Specifiek belicht dit werk de uitdagingen van het integreren van realisme in zowel aanvalstechnieken als verdedigingsstrategieën, zoals het identificeren van haalbare adversariële kwetsbaarheden in malwareclassificatiemodellen of het eerlijk evalueren van de robuustheid van malwaredetectie. Door deze aspecten opnieuw te beoordelen, beogen we een praktischere benadering te bieden voor het versterken van ML-gebaseerde malwaredetectie tegen realistische ontwijkingsaanvallen.

Het onderzoek begint met het verkennen van realistische dreigingsmodellen van ontwijkingsaanvallen en streeft ernaar de grenzen van de huidige kennis en testmethoden op dit gebied te verleggen. Een belangrijke bijdrage aan de aanvalszijde is het voorstellen van een praktische ontwijkingsaanval, genaamd EvadeDroid, die de fundamentele uitdaging aanpakt om ontwijkingstechnieken te ontwikkelen die onafhankelijk zijn van het doelsysteem. Deze onafhankelijkheid is cruciaal, aangezien detectiesystemen in de praktijk vaak functioneren als black boxes die enkel harde labels retourneren. Het proefschrift benadrukt dat adversariële malware niet alleen haar kwaadaardige functionaliteit moet behouden, maar ook zodanig moet worden gemanipuleerd dat deze onopgemerkt blijft tijdens de preprocessering, zodat detectie wordt omzeild. Bovendien is EvadeDroid geoptimaliseerd voor efficiëntie: het minimaliseert het aantal queries en verhoogt de geloofwaardigheid door het injecteren van realistische adversariële payloads, gebaseerd op bestaande code. Hierdoor wordt de aanval praktischer en moeilijker te verdedigen.

Aan de verdedigende kant onderzoekt het proefschrift methoden om efficiënt en effectief de kwetsbaarheden van malwareclassificatiemodellen tegen realistische ontwijkingsaanvallen te identificeren, en bestudeert het robuuste verdedigingsstrategieën om detectiesystemen hiertegen te beschermen. Een belangrijke bijdrage van het defensieve deel van dit onderzoek is het identificeren van haalbare kwetsbaarheden door het genereren van potentiële adversariële malware direct in de kenmerkruimte in plaats van in de probleemruimte, wat efficiëntere en

bredere bescherming biedt. Daarnaast wordt het probleem van spurious correlations onderzocht—misleidende patronen die door malwareclassificatiemodellen ten onrechte kunnen worden geleerd. Door te waarborgen dat de gebruikte kenmerken beter overeenkomen met de daadwerkelijke functionaliteit van malware, kunnen classificatiemodellen worden verbeterd om beter bestand te zijn tegen manipulatie en beter te generaliseren. Ten slotte wordt ingegaan op adversariële training (AT), een gangbare techniek om malwareclassificatiemodellen te versterken tegen adversariële aanvallen. Dit onderzoek specificeert belangrijke dimensies voor het verkennen en evalueren van AT, zoals kenmerkrepresentaties en de realisme van adversariële voorbeelden, door een uniform kader voor te stellen dat helpt bij het identificeren van essentiële factoren die het succes van AT beïnvloeden, zoals modelflexibiliteit. Het proefschrift benadrukt het belang van het beschouwen van al deze factoren in samenhang, in plaats van afzonderlijk, om tot betekenisvolle verbeteringen in robuustheid te komen.

Samengevat biedt dit proefschrift een praktische en toekomstgerichte benadering voor het verdedigen tegen adversariële bedreigingen in malwaredetectie. Het onderstreept de noodzaak van realistischere dreigingsmodellen en betrouwbare verdedigingsstrategieën. Door de complexiteit van praktische ontwijkingsaanvallen aan te pakken en een uitgebreid raamwerk te bieden voor het bouwen van veerkrachtige systemen, draagt dit werk bij aan de ontwikkeling van veiligere en effectievere ML-gebaseerde malwaredetectiesystemen die bestand zijn tegen het steeds veranderende dreigingslandschap. Tot slot worden er enkele veelbelovende onderzoekslijnen voorgesteld, met name gericht op hoe de kwaliteit van trainingsdata de ontwikkeling van robuustere malwareclassificatiemodellen beïnvloedt.

Acknowledgments

Looking back on this long journey, I reflect on the rough and challenging path I've taken to reach this milestone—undoubtedly one of the most significant turning points in my life. The only words that truly come to mind are: "Thank you, God." Thank you for creating me with strength, perseverance, intelligence, passion, hope, and ambition. Thank you for blessing me with devoted and loving parents who raised me with faith and supported me unconditionally. Thank you for allowing me to grow up surrounded by kind and encouraging siblings, whose love and presence have always been a source of comfort. Thank you for placing exceptional teachers, advisors, and mentors along my path. And above all, thank you for the most invaluable gift you have given me—my greatest love, Marzieh, who has dedicated herself fully to this journey alongside me.

Marzieh, you know more than anyone that I could not have made it here without you. I have run, walked, and at times crawled just to keep moving forward, but it was you who kept me standing. Words cannot fully capture the depth of my gratitude for your extraordinary kindness, unwavering support, and boundless dedication. We have walked this path together, and I wholeheartedly dedicate this achievement to you. Still, even this dedication cannot fully reflect all that you have done—for me and for our dear children, Mohammad and Ali, who have been part of this journey every step of the way.

I would also like to extend my heartfelt thanks to my wife's family. I am especially grateful to my late father-in-law—may he rest in peace—whose memory continues to inspire me. My warmest thanks to my mother-in-law for her continued kindness, and to my brothers- and sister-in-law, Ali, Mohammadhadi, and Zeinab, for their sincere interest in my work and for the many engaging conversations we've shared about research.

Before continuing with my acknowledgments, I would like to take a moment to reflect on how I arrived at this point. My story goes back many years to my time as a student. I was always among the top performers in primary and secondary school—a curious boy with endless questions. However, during high school, I drifted somewhat away from being among the top students for various reasons. But as I pursued my bachelor's degree, I returned to my "factory settings" and once again began to thrive academically. A pivotal moment in my academic journey came when I enrolled in a master's course taught by Prof. Mansour Sheikhan. His lectures on artificial neural networks sparked a profound passion for AI research within me. With his mentorship and encouragement, I pursued a novel master's thesis at the intersection of AI and cybersecurity, leading to impactful research and significant achievements—none of which would have been possible without his constant support.

After completing my master's, I continued conducting research with great enthusiasm, driven by the perfectionism I believed was essential for securing a top PhD position. In hindsight, committing to a long-term research project delayed my PhD trajectory by nearly five years. However, that period of exploration may have been a necessary detour, ultimately guiding me toward a PhD position focused on machine learning security in the malware

domain at one of top universities in the Netherlands. I'm deeply grateful to Veelasha and Erik for believing in me and offering me the position after our first interview.

I began my PhD journey during the height of the COVID-19 pandemic—a time of heightened challenges. Moving from Iran to the Netherlands was far from easy; the university was nearly deserted, with most working remotely, and even Veelasha had already moved on to RUB. I still recall my first day at Mercator 1, where Irma and Joan welcomed me warmly. Despite being physically distant, Veelasha made substantial effort to prepare me for my PhD, offering invaluable guidance on university life and adjusting to life in the Netherlands. Though remote supervision came with its challenges—for both of us—we found our rhythm and made steady progress together. Veelasha, thank you for trusting in me and for your ongoing support and mentorship. I am sincerely grateful for the way you helped me grow, both as a researcher and as a person, particularly in areas like academic networking, which I now recognize as vital in research. Likewise, I extend my deep gratitude to Erik for facilitating the logistical aspects of my work, providing thoughtful feedback, and offering timely assistance. Erik, your trust and support have been instrumental in helping me find my footing throughout this journey.

I also appreciate Zhengyu and Zhuoran for their collaboration and engaging discussions, which significantly deepened my understanding of adversarial machine learning. Likewise, I am grateful to Lorenzo and Fabio for welcoming me into their research group in the UK, where I had the opportunity to explore an exciting project. Similarly, I truly valued my collaboration with Jacopo and Daniel during my research visit, especially working closely with Jacopo on a project that was highly relevant to both our PhD research.

A special thanks to the Iranian community in Nijmegen—Ahmad, Arash, Hamidreza, Farzaneh, Parisa (Naseri), Maryam, Negin, Hadi, Sadegh, Masoud, Akbar, Raheleh, Esmail, and Arman—for their unwavering support of me and my family throughout this journey. Without them, settling in Nijmegen, especially for my family, would have been far more difficult. I am also deeply grateful to Alireza for his generosity in sharing his experience, guiding me through the hiring process, and helping me navigate the many bureaucratic steps of relocating to the Netherlands.

Many thanks to Behnam, Omid, and Parisa (Amiri Eliasi) for their friendship and support, which made my time at work more enjoyable. I'm also grateful to Twan and Pol for the opportunity to assist in their course—the experiences that helped me grow as an educator.

I extend my appreciation to all my colleagues in the Digital Security and Data Science groups—Joan, Gunes, Irma, Janet, Shanley, Ronny, Ileana, Martha, Faegheh, Gijs, Alex, Nik, Azade, Behrad, Vahid, Yanis, Krijn, and many others, especially my office mates Zahra, Paulus, Charlotte, and Alexandre—for their cooperation and friendship.

Finally, I am sincerely grateful to the members of the manuscript committee—Lejla, Konrad, Mauro, Juan, and Jason—for taking the time to evaluate my dissertation and grant their approval. I would also like to thank Stjepan for joining the examination board and being part of this important milestone. This journey has been long and challenging, yet enriched by growth, learning, and meaningful connections. I am grateful for every step and for those who have walked it with me.

Hamid Bostani
Nijmegen, The Netherlands—June 2025

1

Introduction

In today's cybersecurity landscape, *mal*icious soft*ware* (malware) poses a growing and evolving threat. Such software is intentionally designed to harm victim machines, such as computers or smartphones [1]. Malware uses various techniques, both simple and sophisticated, that can cause serious and sometimes irreversible damage to systems. In the ongoing battle against such threats, Machine Learning (ML) has emerged as a promising approach in enhancing malware detection [2]. ML can be employed in static analysis to inspect software source codes for detecting malicious content, or in dynamic analysis to monitor the behavior of running software to identify malicious activities. For instance, leading Android application stores, such as Google Play, leverage ML in both static and dynamic analysis to identify potentially harmful applications [3]. Utilizing ML for malware detection is valuable, as system security requires increasingly intelligent decision-making to keep up with malicious actors who are constantly advancing their attack methods. Nevertheless, the trustworthiness of malware classifiers[1] remains a major concern for both academia and industry [4]. A key pillar of building a trustworthy malware classifier is ensuring its robustness, which enables the detector to perform reliably even under challenging conditions during its inference phase, such as noisy representations caused by errors in feature extraction. Achieving robustness, especially in malware detection, which is adversarial by nature [5], presents significant challenges as ML systems can be targeted by adversarial attacks, which specifically aim to diminish their robustness. Evasion attacks are among the hazardous forms of adversarial attacks which entail crafting adversarial input instances named adversarial examples (AEs) to circumvent ML systems. In the context of malware, adversaries alter the static features of malware (e.g., control flow graph) to generate *adversarial malware*, a form of AE that not only evades malware classifiers but also has the potential to compromise victim machines. Such type of adversarial attack must ensure the malware's inherent properties, such as its malicious functionality, remain intact after adversarial manipulations.

1.1 Revisiting Realism for Adversarial Malware

To improve the adversarial robustness[2] of malware classifiers, it is essential to consider realism, referring to threat models and defenses that align with real-world constraints. This

[1] In this dissertation, the terms malware classifiers and ML-based malware detectors are used interchangeably.

[2] In the context of this dissertation, adversarial robustness specifically refers to the robustness of ML systems during the inference phase against evasion attacks.

1

ensures that solutions go beyond theory and are applicable in practical settings. Realism is a crucial aspect for both attackers and defenders in Adversarial Machine Learning (AML), yet it has been underemphasized in the malware domain because of the unique properties of malware, which set it apart from other entities such as images. While AML typically focuses on identifying and mitigating vulnerabilities within ML systems [6], the challenges of securing ML systems in the malware domain are significantly more pronounced compared to fields like computer vision. This is primarily due to the mismatch between the *problem space* and the *feature space*. The problem space refers to real-world representations of malware or benign software—such as actual Android apps or Windows executables—while the feature space represents these samples in abstract forms, typically as binary or numerical vectors. This mismatch introduces several domain-specific characteristics that make securing malware classifiers particularly difficult:

- **Non-standardized Features.** Malware detection lacks standardized feature definitions [7], unlike other domains such as computer vision, where fixed features like pixels are used. This flexibility allows attackers to target only key features relevant to specific detectors, whereas, in computer vision, all pixels are involved. This also gives adversaries more opportunities to exploit vulnerabilities in feature representations. For instance, if a feature representation is not expressive enough to differentiate malware from goodware [8], attackers can alter malware so that its representation mimics that of benign programs, without altering the malware's original functionality. Finally, non-standardized features make it harder for defenders to create generalized defenses, as for each detector, defenses must be tailored to the specific features used by each detection system. For example, if detectors rely on permissions, defenders must assess how these can be adversarially altered, whereas those based on API calls require evaluating different manipulation methods.

- **Structured Program Files and Discrete Feature Space.** Program files are highly structured and often represented by discrete features, making it difficult to alter them without rendering the files non-executable or harmless [7]. However, in the malware domain, since the goal is not only to evade detection but also to successfully compromise victim machines, any modifications must adhere to domain constraints (e.g., preserving the semantics of malware after adversarial manipulations), ensuring the malicious programs remain functional and harmful. In fact, satisfying these constraints ensures that the malware's malicious functionality remains intact, as any changes made to evade detection must not interfere with its ability to carry out its intended harmful actions.

- **Non-Invertible and Non-Differentiable Feature Mapping.** Since adversarial perturbations typically occur in the feature space by perturbing the representation of malware, attackers must reconstruct the malware based on the perturbed feature representation. However, the mapping function from the problem space to the feature space in malware detection is non-invertible [9], meaning it is not possible to reconstruct a functional program file based solely on its feature representation. This is because the feature representation simplifies the actual code, capturing only aspects relevant for detection, not the full logic or structure of a program. Additionally, this mapping function is non-differentiable [9], complicating the use of gradient-based evasion attacks [10].

> Program files are structured and non-numerical; therefore, converting them into numerical features for ML models acts like a non-differentiable layer. Specifically, transforming a feature representation generated by a gradient-based attack into a functional program file presents a challenge, as back-propagating the loss gradient through this mapping is not feasible [10].

These characteristics intensify the challenges encountered by attackers and defenders in the malware domain. Particularly, the practical effectiveness of many proposed evasion attacks and defense strategies remains questionable, highlighting the need for a deeper focus on realistic threat models and reliable defenses that reflect the nature of the malware domain.

Over the past decade, significant research has been dedicated to exploring the vulnerabilities of malware classifiers through the investigation of evasion attacks. However, the real-world applicability of the proposed adversarial attacks remains uncertain. Many studies [9, 11–25] assume that attackers have in-depth knowledge of the target classifiers used for malware detection, whereas, in practice, malicious actors often have limited access to such information. Although some studies [16, 26–28] acknowledge this limitation by investigating evasion attacks in Zero-Knowledge (ZK) settings, they frequently neglect the evasion costs (e.g., the number of queries required to target malware classifiers), which represent important real-world constraints. Moreover, the majority of studies [9, 11–28] generate AEs through manipulations in the feature space. However, the realizability of adversarial programs in the problem space has received less attention due to the inverse feature-mapping problem or the failure to meet domain constraints that ensure feasible programs.

On the other hand, many recently proposed defenses [12, 14, 15, 17–19, 24–26, 29–38] may not adequately protect malware classifiers from evasion attacks. This inadequacy is either due to the lack of clarity regarding their focus on feasible, vulnerable regions of the ML decision space or their failure to evaluate against strong and realistic evasion attacks, which generate feasible adversarial malware by adhering to domain constraints. Furthermore, malware classifiers are more prone to spurious correlations—misleading associations between input features and target labels—due to malware's dynamic nature, imbalanced datasets, and noisy, high-dimensional features, which lead models to rely on superficial patterns rather than meaningful ones. Current defenders fail to account for the intrinsic tendency of malware classifiers to learn spurious correlations that significantly diminish the performance of ML systems in adversarial settings. Finally, defenders often lack a clear understanding of how factors related to key aspects of malware classifiers, such as feature representation and learning algorithms (e.g., feature space dimensionality and classifier flexibility), contribute to the vulnerabilities of ML models to evasion attacks. This knowledge gap makes it difficult to fully grasp how these factors influence the models' adversarial robustness, which is crucial for effective defense against such attacks.

1.2 Problem Statement

ML has gained prominence in malware detection because it not only overcomes the limitations of traditional methods, such as evading signature-based detectors with repacking or polymorphism, but also excels at detecting evolving, unseen malware through model generalization and knowledge adaptation [7]. ML is typically employed in either static analysis, which distinguishes malware from goodware based solely on source code, or

(a) Reactive Arms Race

(b) Proactive Arms Race

Figure 1.1: A conceptual depiction of the arms race in AML [39], showing two perspectives.

dynamic analysis, which evaluates runtime behavior. Although malware classifiers have demonstrated high performance in various studies, they remain vulnerable to adversarial attacks, particularly evasion attacks that alter malware to mislead malware classifiers during their inference phase.

To improve the security of malware classifiers against evasion attacks, defense strategies follow two approaches, namely security by obscurity and security by design [39]. Security by obscurity relies on hiding system details from adversaries, while adversaries could potentially uncover these obscured details. This approach results in a reactive arms race between adversaries and defenders. As shown in Figure 1.1a, in a reactive arms race, adversaries examine the ML systems to identify and exploit vulnerabilities to craft evasion attacks. Defenders (i.e., classifier designers), in turn, respond by assessing these threats and updating the ML systems accordingly [39, 40]. Conversely, as shown in Figure 1.1b, the security-by-design approach embodies a proactive arms race, advocating for the development of ML systems that are inherently secure against evasion attacks from the outset. In this approach, defenders (i.e., classifier designers) anticipate potential threats by forecasting evasion attacks and then fortify ML systems by implementing robust countermeasures designed to be secure from the outset [39, 40]. This dissertation endorses a security-by-design approach to safeguard malware classifiers against evasion attacks by (i) anticipating potential evasion attacks and analyzing the vulnerabilities of malware classifiers to these adversarial attacks, and (ii) developing defenses that enhance the adversarial robustness of malware classifiers. Indeed, it begins by examining realistic evasion attacks and feasible adversarial perturbations to reveal the vulnerabilities of ML models. Subsequently, it investigates hardening solutions to strengthen malware classifiers against realistic evasion attacks. It is worth noting that we focus on exploring evasion attacks targeting ML-based malware detection systems that rely on static analysis due to their scalability and resource efficiency, which enable large-scale evaluations. Below, we first motivate the environment considered in this dissertation for our investigations and then present the research questions that it aims to explore.

1.2.1 Target Environment

Malware comes in various forms, such as Windows Portable Executables, PDFs, and JavaScripts. This thesis focuses specifically on Android malware, as Android provides an ideal environment for research due to its open-source nature, large user base, and access to extensive datasets including large collections of timestamped Android Packages (APKs). This open-source platform allows a wide range of developers to create applications, which can lead to inconsistent security practices across the ecosystem [41]. The inconsistency in security practices among Android applications leads to exploitable vulnerabilities, making the platform a compelling subject for malware research. Additionally, Android's dominance

in the global smartphone market—with over 70% market share [42]—subjects a vast user base to increased malware risks. The ability to sideload apps from third-party sources, coupled with diverse security measures across various Android devices, further escalates its susceptibility to security threats [43]. These factors collectively make Android a particularly interesting platform for malware research. Finally, repositories like AndroZoo [44] offer researchers access to millions of timestamped APKs, which is invaluable for large-scale studies and ensuring the accuracy and reliability of malware detection systems.

Although the primary focus of our research is on Android malware, the insights and challenges explored in this dissertation extend beyond the Android ecosystem. Challenges such as feature representations in malware classifiers, realistic evasion attacks, identifying ML vulnerabilities, the tendency of malware classifiers to learn spurious correlations, and the uncertain effectiveness of adversarial training are common issues that affect malware detection across various platforms. These common challenges highlight the broader relevance of this research to the general field of malware defense.

1.2.2 Practical Evasion attacks

In real-world scenarios, adversaries often lack access to detailed information about target malware classifiers as antivirus systems typically function as black-box models that can only be queried externally [45]. For instance, adversaries are often limited to performing hard-label attacks by querying the target detectors, where they can only access the classification labels assigned to the input samples by target malware classifiers [46], rather than the more detailed confidence scores, which would be more advantageous for generating AEs. Efficient querying is also essential, given the costs associated with each query and the risk of detectors blocking suspicious activities [45]. Furthermore, adversaries must ensure that the adversarial malware they generate remains functional [47], allowing it to be installed on victim machines and execute malicious actions after successfully bypassing malware detection systems. The first research question of this thesis explores a practical evasion attack that meets the requirements of these real-world constraints, addressing the challenge of generating adversarial malware that can both evade detection and maintain its malicious functionality.

> **Q1**. What threat model is truly practical for evasion attacks targeting malware classifiers in real-world scenarios?

1.2.3 Effectiveness of Defenses

Realistic evasion attacks must generate functional adversarial malware in the problem space, meaning they must adhere to domain constraints in the problem space [9]. For instance, adversarial manipulations must be robust against non-ML preprocessing techniques (e.g., code pruning, which minimizes code size by eliminating dead or unreachable code), as malware classifiers often preprocess input programs before classification to exclude unused payloads, such as unused permissions in Android apps [9]. AEs that are not robust to preprocessing may be filtered out by the malware classifiers, rendering the attack unsuccessful. In the feature space where malware classifiers operate, not all feature representations are feasible, as they must correspond to functional and valid programs in the problem space. As depicted in Figure 1.2, realistic evasion attacks are restricted to targeting specific regions in the blind

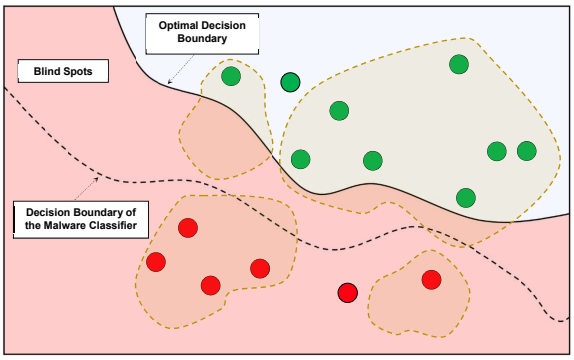


Figure 1.2: A conceptual illustration of feasible regions (outlined by dashed orange closed lines) in the feature space, where malware (red dots) and goodware (green dots) can be positioned. The blind spot regions within these closed lines are feasible vulnerable areas that can be targeted by realistic evasion attacks. The red and green dots located outside the dashed orange boundaries represent unrealizable malicious and benign programs, respectively.

spots of the feature space where these feasible representations exist. It is important to note that blind spots refer to the areas between the decision boundary learned by a malware classifier and the ideal potential decision boundary, where the classifier fails to correctly identify malware.

To effectively defend malware classifiers against such attacks, defenders should concentrate on protecting these vulnerable regions. Identifying these regions by generating realizable AEs in the problem space—those that meet domain constraints—can be useful, though this process is both computationally intensive and time-consuming. Furthermore, realistic evasion attacks used by defenders to generate realizable AEs are based on limited problem-space transformations[3], potentially lacking other unseen problem-space transformations. As a result, defenders may struggle to protect malware classifiers against attacks that use alternative, undiscovered transformations to generate AEs, since these problem-space transformations target unknown vulnerable regions. The next research question explores a technique to quickly identify these vulnerable areas within the decision space of malware classifiers, enabling more robust defenses against realistic evasion attacks without revealing the full extent of ML vulnerabilities in the problem space.

Q2. How can vulnerable regions in malware classifiers be efficiently identified to defend against realistic evasion attacks?

1.2.4 Spurious Correlations in Malware Classifiers

In the context of systems security, spurious correlations refer to patterns in data, such as relationships between input features and the target label, that are unrelated to the actual security problem, such as malware detection, but create shortcuts for classification [48]. For example, if a specific permission appears frequently in malware apps, an ML model might

[3]Problem-space transformations include manipulations like injecting dead code to modify software.

learn to rely on the presence of that permission to classify malware, instead of identifying genuine malicious behavior. Robust ML systems emphasize learning robust features—those directly linked to the classification task and less vulnerable to adversarial perturbations—over unrelated patterns in the data. Altering these robust features carries a high risk of changing the semantics of the input programs (e.g., changing the malware functionality), which is undesirable for evasion attacks. The next research question highlights the impact of spurious correlations on the adversarial robustness of malware classifiers, specifically investigating how the influence of irrelevant patterns can be eliminated to enhance the classifier's resilience against realistic evasion attacks.

> **Q3**. How can the impact of spurious correlations be reduced to secure malware classifiers against realistic evasion attacks?

1.2.5 Influential Factors for Malware Classifiers

There are numerous studies conducted in the malware domain to shield malware classifiers against evasion attacks, e.g., defensive distillation [17], weight bounding [16, 28], and monotonic classification [49]. Adversarial Training (AT) [50] emerges as the most successful defense strategy against evasion attacks [51, 52] which leverage the usage of AEs during the training phase to mitigate the generalizability concerns inherent in ML models. To this end, an effective mechanism for AT formulates the training procedure as a robust optimization via a min-max optimization problem, where the inner maximization problem seeks optimal adversarial perturbations, and the outer minimization problem optimizes model parameters [53]. This proactive defense, which is also known as adversarially robust training [52], enhances the generalizability of ML models in classifying unseen samples, such as AEs, by revealing blind spots in their decision space. To advance AT in the context of malware detection, it is crucial to provide an in-depth understanding of AT's capability not only in uncovering blind spots but also in maintaining the clean performance of ML models. This requires exploring various factors that influence AT while ensuring evaluations are well-aligned with real-world scenarios to accurately assess improvements in adversarial robustness. The next research question addresses this gap by investigating the intertwined roles of different factors in strengthening malware classifiers through AT.

> **Q4**. What are the key factors influencing adversarial training, and how do they affect its effectiveness in malware detection?

1.3 Structure of Dissertation

This dissertation believes we need to rethink the security of ML used in malware detection by revisiting realism. Throughout the chapters, we investigate evasion attacks and defenses with a focus on realism, highlighting practical threat models of evasion attacks in real-world scenarios, and examining effective defensive strategies to strengthen malware classifiers against such attacks. Our exploration opens with **Chapter 2** to lay the groundwork for our examination of the adversarial robustness of malware classifiers. Specifically, this chapter provides a detailed foundation for understanding malware detection in the context of AML.

It begins by outlining the fundamentals of malware detection and then discusses how evasion attacks exploit the vulnerabilities of malware classifiers to generate adversarial malware aimed at bypassing detection. The chapter also delves into how different threat models help simulate realistic evasion attacks, highlighting the potential risks that malware classifiers face. Furthermore, it reviews key defense mechanisms aimed at enhancing the robustness of malware classifiers, ensuring they can withstand evasion attacks. Reviewing these topics is essential for this thesis, as it establishes the necessary context for investigating adversarial robustness and contributes to the development of more secure ML-based malware detection systems.

Since developing malware classifiers that are secure against evasion attacks requires an understanding of ML vulnerabilities to evasion attacks, the first part of the thesis, covered in **Chapters 3 and 4**, investigates practical evasion attacks and the susceptibility of ML models to these threats within the feature space. Specifically, **Chapter 3** presents a novel evasion attack method designed to evade Android malware classifiers in real-world scenarios. The proposed attack, which is named EvadeDroid, is a practical evasion attack that operates under ZK settings in the problem space. By utilizing a random search optimization algorithm, EvadeDroid iteratively applies problem-space transformations derived from benign apps to bypass malware classifiers. This optimization approach effectively selects and injects suitable transformations by relying on feedback from the target classifier. EvadeDroid not only operates without needing any knowledge of the malware classifier's internal workings but also achieves high query efficiency with minimal interactions. As a result, EvadeDroid achieves high evasion rates while preserving the malware's original functionality, demonstrating its feasibility in real-world applications. Furthermore, **Chapter 4** introduces a novel approach to identify regions vulnerable to realistic evasion attacks by exploring realizable AEs within the feature space. This approach ensures that the AEs adhere to domain constraints essential for feasible Android apps. To this end, we interpret these domain constraints as meaningful feature dependencies and propose a method to learn them through feature correlations. Leveraging these learned constraints, our approach enhances detection by effectively distinguishing AEs from feasible apps. Moreover, it addresses the shortcomings of prior studies, particularly in maintaining realism and efficiency in adversarial hardening. By integrating these constraints into AT, we significantly strengthen the robustness of Android malware detection against realistic evasion attacks.

The second part of the thesis, presented in **Chapters 5 and 6**, examines the defenses that protect malware classifiers against the ML vulnerabilities discussed in the first part. ML models used for malware detection often grapple with the challenge of spurious correlations, which can severely hinder their performance on out-of-distribution data, such as AEs. **Chapter 5** presents a novel approach to addressing this vulnerability through a domain adaptation technique that enhances the generalizability of malware classifiers. By establishing a robust feature space that aligns the distributions of malware and adversarial malware, our method mitigates the influence of misleading features. This alignment allows classifiers to focus on genuine semantic patterns associated with software functionality, thereby bolstering their resilience against realistic evasion attacks. **Chapter 6** investigates AT as the most effective strategy for enhancing malware detection. This chapter highlights that the effectiveness of AT in hardening malware classifiers against realistic evasion attacks has not been thoroughly explored. Many studies have either not conducted a comprehensive

exploration of the full space or have used inappropriate evasion attacks (e.g., unrealistic or weak ones), limiting the validity of their conclusions. This chapter proposes a unified framework to investigate the impact of various influential factors—within data, feature representations, classifiers, and robust optimization settings—on the effectiveness of AT, using different evaluation criteria. It systematically explores the roles of these factors in strengthening malware classifiers, identifies evaluation pitfalls that may lead to misleading conclusions in state-of-the-art approaches, and offers key insights to guide future research toward optimal AT configurations.

In **Chapter 7**, we conclude the thesis by summarizing the key contributions of each chapter and reflecting on how they address the research questions outlined in Section 1.2. We revisit the findings from our exploration of evasion attacks, defenses, and practical threat models in malware detection, highlighting the advancements made in understanding and improving the adversarial robustness of ML models in this domain. Finally, we close the dissertation by discussing potential directions for future research, focusing on areas where further investigation could enhance the security of ML-based malware detection systems. In particular, we underscore the significance of exploring evasion attacks targeted at malware classifiers used in dynamic analysis, as well as analyzing the influence of malware types and families on the effectiveness of evasion attacks in shaping adversarial malware. Furthermore, we caution against the growing threat of malware propagation through malicious public repositories. Specifically, we stress the importance of proactively analyzing evasion attacks that bypass detection mechanisms intended to identify and mitigate these threats. Our final recommendation is to provide a detailed discussion on how future research should concentrate on improving the robustness of malware detection by leveraging high-quality, in-distribution data prepared by advanced techniques, which may lead to the development of malware classifiers that are more resilient to both in-distribution and out-of-distribution data, such as adversarial malware.

1.4 List of Publications

I have published the following papers during my Ph.D. at Radboud University.

1.4.1 Main Publications Contributing to Thesis Chapters

This dissertation primarily draws on five publications, most of which were collaborative efforts with other authors. All chapters have either been published or are currently under review in peer-reviewed journals and conferences. My specific contributions to each chapter, based on the published works, are detailed in the following list.

- **Chapter 3.** Subverting Machine Learning in Malware Detection.

  This chapter is based on *EvadeDroid: A Practical Evasion Attack on Machine Learning for Black-Box Android Malware Detection*[4], by **Hamid Bostani** and Veelasha Moonsamy, which was published in *Computers & Security* in 2024. In this research, I led the project, overseeing the conceptualization, design, and empirical evaluation of EvadeDroid, a new evasion attack proposed to deceive ML-based Android malware detection. My extensive literature review, which examined prior studies on evasion

[4] https://doi.org/10.1016/j.cose.2023.103676

attacks and their roles in malware detection, informed the development of EvadeDroid and helped identify limitations in existing evasion attacks. I was responsible for drafting the initial manuscript, incorporating insights from existing literature to effectively position our work within the wider landscape of AML. Additionally, I handled the implementation of the proposed attack and the reproduction of existing evasion strategies and malware detectors to empirically validate the effectiveness of EvadeDroid. Veelasha Moonsamy contributed valuable feedback and revisions, which were instrumental in enhancing the clarity and quality of the paper.

- **Chapter 4.** Exposing Vulnerabilities in Machine Learning for Malware Detection.

  This chapter is based on *Level Up with ML Vulnerability Identification: Leveraging Domain Constraints in Feature Space for Robust Android Malware Detection*[5], by **Hamid Bostani**, Zhengyu Zhao, Zhuoran Liu, and Veelasha Moonsamy, which was published in *ACM Transactions on Privacy and Security* in 2025. In this work, I took on the role of lead author, driving the conceptualization, design, and empirical evaluation of our method for detecting adversarial vulnerabilities in malware classifiers. I conducted a thorough literature review, focusing on AML and the application of domain constraints in the feature space, which directly informed our approach for strengthening malware classifiers. I wrote the initial draft of the paper, synthesizing prior research to place our method within the broader discourse on vulnerability detection in ML models. My responsibilities also included implementing the full source code and carrying out data collection and preparation. Zhengyu Zhao contributed to the conceptual development of the study and, along with Zhuoran Liu and Veelasha Moonsamy, provided insightful feedback and edits that helped refine the manuscript, enhancing its clarity and precision. Their contributions were crucial to ensuring the overall quality of the paper.

- **Chapter 5.** Enhancing Adversarial Robustness with Robust Feature Space.
  This chapter is based on *Improving Adversarial Robustness in Android Malware Detection by Reducing the Impact of Spurious Correlations*[6], by **Hamid Bostani**, Zhengyu Zhao, and Veelasha Moonsamy, which was presented at the *29th European Symposium on Research in Computer Security International Workshops* (ESORICS 2024 International Workshops) in 2024. As the lead author of this study, I was responsible for the overall conceptualization, design, and empirical evaluation of a method aimed at improving adversarial robustness in Android malware detection by addressing the impact of spurious correlations. My literature review explored existing research on adversarial robustness and feature representations, which informed the development of our approach to enhance model resilience. I wrote the initial draft of the paper, integrating findings from related works to frame our method within the larger context of adversarial defense strategies. I also implemented the entire source code and managed the data collection and preparation. The valuable feedback and revisions provided by Zhengyu Zhao and Veelasha Moonsamy significantly contributed to improving the clarity and quality of the final manuscript. Their contributions ensured the paper's strength and coherence.

[5] https://doi.org/10.1145/3711899
[6] https://doi.org/10.1007/978-3-031-82362-6_13

- **Chapter 6.** Enhancing Adversarial Robustness with Robust Optimization.
  This chapter is based on *On the Effectiveness of Adversarial Training on Malware Classifiers*[7], by **Hamid Bostani**, Jacopo Cortellazzi, Daniel Arp, Fabio Pierazzi, Veelasha Moonsamy, and Lorenzo Cavallaro, which has been under peer review since 2025. In this study, I led the project, guiding the conceptualization, literature review, and formulation of key research questions and hypotheses regarding the effectiveness of AT for malware classifiers. My literature review focused on existing works concerning AT and vulnerabilities in malware detection, which helped define the research questions related to perturbation budgets, confidence levels of AEs, and the impact of feature representations on AT performance. I authored the majority of the paper, writing key sections such as the introduction, related work, methodology, and evaluation, with a particular focus on pitfalls, robustness optimization settings (perturbation bounds, confidence levels of AEs, and domain constraints), and key findings. I was also responsible for implementing the source code, including the development of classifiers (DNNs[8] and linear SVM[9]), feature-space and problem-space attacks (PGD[10], JSMA[11], and EvadeDroid), as well as the AT method. Additionally, I designed analysis tools for evaluating feature importance and plotting performance metrics. Jacopo Cortellazzi was a co-contributor in brainstorming, data preparation, implementation, and authoring. Daniel Arp, Fabio Pierazzi, Veelasha Moonsamy, and Lorenzo Cavallaro contributed to brainstorming and provided essential feedback and edits, which significantly improved the paper's clarity and coherence. Jacopo Cortellazzi, Fabio Pierazzi, and Lorenzo Cavallaro also contributed to the conceptual development of the study.

- **Chapter 7.** Conclusions and Outlook.
  The outlook section of this chapter is based on *Beyond Learning Algorithms: The Crucial Role of Data in Robust Malware Detection*, by **Hamid Bostani** and Veelasha Moonsamy, accepted for publication in *IEEE Security & Privacy* in 2025. In this work, I was the lead author, responsible for the conceptualization and literature review concerning the use of coreset methods to improve adversarial robustness in malware detection. My review of existing research on coreset techniques and adversarial robustness provided a foundation for developing arguments and insights into the potential opportunities and challenges of applying coreset methods in this domain. I wrote the entire paper, synthesizing insights from previous studies to build a clear argument on the critical role of data quality in enhancing the adversarial robustness of malware classifiers, and how coreset techniques can help overcome challenges in achieving robust malware detection. Throughout the writing process, Veelasha Moonsamy provided valuable feedback, which improved the manuscript's clarity and overall quality.

[7] `https://arxiv.org/abs/2412.18218`
[8] Deep Neural Networks
[9] Support Vector Machine
[10] Projected Gradient Descent
[11] Jacobian-based Saliency Map Attack

1

1.4.2 Other Publications

I have also contributed to the following paper during my Ph.D. at Radboud University; however, it is not part of this dissertation.

- *Targeted and Troublesome: Tracking and Advertising on Children's Websites*[12], by Zahra Moti, Asuman Senol, **Hamid Bostani**, Frederik J. Zuiderveen Borgesius, Veelasha Moonsamy, Arunesh Mathur, and Gunes Acar, which was presented at the *45th IEEE Symposium on Security and Privacy* (IEEE S&P) in 2024. In this work, I contributed to brainstorming the research and collaborating on preparing the list of child-directed websites by conducting a literature review on web classification, conceptualization, validating various potential techniques, and manually labeling websites. I also implemented scripts for online labeling of the collected websites using VirusTotal, analyzed the results, verified the implemented LLM-based method for web classification. Additionally, I contributed to preparing the initial structure of the paper and co-authored the sections related to compiling the list of child-directed websites.

1.5 Code and Data Management

The source code developed and datasets collected for the research in various chapters of this thesis are listed below:

- Chapter 3: *EvadeDroid* developed by **Hamid Bostani**.
  https://github.com/HamidBostani2021/EvadeDroid.
- Chapter 4: *Robust Android Malware Detector* develped by **Hamid Bostani**.
  https://github.com/HamidBostani2021/robust-Android-malware-detector.
- Chapter 5: *Robust Feature Space* developed by **Hamid Bostani**.
  https://github.com/HamidBostani2021/robust-feature-space.
- Chapter 6: *Robust Optimization for Malware Detection* developed by **Hamid Bostani** and Jacopo Cortellazzi.
  https://github.com/HamidBostani2021/robust-optimization-malware-detection.

The supplementary materials used in all chapters of this thesis are written in Python. Java is also used in the development of EvadeDroid, introduced in Chapter 3. In Chapters 3 to 6, we used the dataset and reproduced the problem-space evasion attack released in the S2Lab repository[13]. However, access to these materials requires permission from the owners. The dataset[14] prepared by Zhang et al. [54] was also used in Chapter 6.

[12] https://ieeexplore.ieee.org/abstract/document/10646733
[13] https://s2lab.cs.ucl.ac.uk/projects/intriguing/
[14] https://github.com/seclab-fudan/APIGraph

2 Background

As the cybersecurity landscape continues to evolve, malware detection systems face growing challenges from adversarial threats. This chapter provides the foundational knowledge necessary for understanding both malware detection and Adversarial Machine Learning (AML) in this context. It begins by outlining the principles of malware detection and the role of ML in identifying malicious software. The discussion then shifts to the security concerns surrounding ML-based malware detection, focusing on AML, with particular attention to evasion attacks—the adversarial threat examined in this dissertation. Then, we review various threat models, which offer insights into the risks malware classifiers face, and defense mechanisms aimed at enhancing the robustness of malware classifiers against adversarial manipulation. By presenting these essential concepts, this chapter establishes the groundwork for investigating the security of malware classifiers against realistic evasion attacks in subsequent chapters.

2.1 Overview of Malware Detection

Malware is any software deliberately designed to disrupt, damage, or gain unauthorized access to victim machines, including computer systems or networks [2]. Its effects include data theft, data loss, and damage to hardware and software [55]. Malware appears in various forms, such as Windows or Android malware, but their objectives are similar within their respective categories. For example, trojans trick users by posing as legitimate software, viruses spread between devices, spyware gathers sensitive information without consent, and ransomware locks systems or files, demanding payment for access [2].

Malware continues to be a major global cybersecurity threat. With over a billion malware programs in existence and hundreds of thousands of new instances discovered daily [56], the need for malware detection systems is more critical than ever. These systems are essential for mitigating the widespread impact of malware and protecting users and systems from ongoing threats.

2.1.1 Malware Detection

Malware detection is a crucial analytical process designed to determine whether software, such as an Android app, is intended to carry out malicious activities on a target system, like Android devices. This process involves examining all executable programs on the system to distinguish malware from benign software. Traditional malware detection techniques often

rely on signature-based methods, where malware signatures are extracted by malware analysts from malicious programs [57]. However, these systems have notable limitations. For instance, signature extraction is labor-intensive and can be circumvented through various techniques such as encryption, repacking, and polymorphism [57]. In contrast, many advanced malware detection approaches utilize behavior-based methods. These techniques analyze the behavior of suspicious programs to determine whether they are malicious [58]. Behavior-based detectors typically evaluate programs in isolated environments (e.g., sandboxes) before allowing them to execute in real-world settings. Nonetheless, these systems have their own limitations, such as a tendency to produce high false-positive rates [58] or an inclination for malware to conceal its malicious functionality during evaluation [59]. In practice, malware detection operates as a pipeline, where each stage utilizes different malware analysis techniques to assess software. The goal of malware analysis is to uncover the functionality, origin, and potential impact of software, providing insight into its behavior [2]. Malware analysis can be broadly divided into static and dynamic analysis [60]:

- Static analysis involves classifying software based solely on their code or binary structure (i.e., raw bytes), without requiring their execution. This approach relies on either code-level features such as strings, API calls, and control flow graphs [2], or binary-level features, such as byte sequences [61], to identify potentially malicious behavior.

- Dynamic analysis detects malware by observing its behavior during execution, such as time-dependent sequences of system calls. This approach leverages runtime characteristics, including API call traces and memory usage patterns, to identify malicious activity [2]. Endpoint Detection and Response (EDR) is a widely recognized dynamic analysis technology designed to continuously monitor and analyze endpoint activities in real-time, enabling rapid responses to threats on devices such as computers and mobile phones [62].

2.1.2 Machine Learning in Malware Detection

ML is a vital branch of Artificial Intelligence (AI) that enables systems to learn from prior collected data—referred to as experience—without requiring explicit programming [63]. Nowadays, ML has become a fundamental computational approach in data processing, finding applications across various domains, such as image processing and speech recognition. Over the past years, ML has been widely used by cybersecurity researchers for malware detection to achieve effective solutions, e.g., detecting zero-day malware. *Supervised learning* is the predominant ML approach for malware detection. This technique utilizes labeled samples to build a binary classifier capable of distinguishing malware from goodware. More recently, Large Language Models (LLMs) have emerged as a promising ML technique in this domain. LLMs, which can offer zero-shot reasoning in malware detection, are trained on vast pre-trained knowledge (e.g., large amounts of code) instead of specifically labeled datasets [64]. These models have demonstrated strong capabilities in static analysis tasks—such as software bug detection [65]—which are closely related to malware detection, further highlighting their potential in this domain.

Building an effective malware classifier with supervised learning requires automating workflows through an ML pipeline. This pipeline comprises several stages, including *data*

preparation (data collection, cleaning, and feature engineering), *model training*, and *model deployment*. During model training, the classifier learns from labeled data to optimize its performance in identifying malware and benign samples, using metrics like accuracy or expected loss. In the model deployment phase, the trained classifier is used to identify malware in real-world scenarios. By automating these stages through a well-structured ML pipeline [66], researchers and developers can streamline the creation and deployment of effective malware detection systems. However, the success of such systems relies on addressing key challenges, such as ensuring high-quality training data, adapting to the ever-evolving nature of malware, and strengthening the model's resilience to evasion attacks, which is the primary focus of this dissertation.

2.2 Adversarial Susceptibility of Malware Classifiers

ML techniques used for malware detection are data-driven approaches that can be significantly impacted by deliberate data manipulation carried out through adversarial attacks [67]. These type of attacks aim to cause misclassification by exploiting vulnerabilities in the ML pipeline at either the training or inference phase. Specifically, based on the attack time, adversarial attacks can be classified into two main categories:

- Poisoning attacks (also referred to as causative attacks [68]): These attacks involve injecting intentionally crafted data points into the training dataset by modifying labels or features to corrupt the training process and compromise the resulting model [69]. Large-scale poisoning attacks can be automated to craft highly effective poisoned samples, allowing attackers to bypass security mechanisms at scale [70].

- Evasion attacks (also known as exploratory attacks [68]): These attacks target the model during the inference phase, attempting to carefully craft adversarial examples (AEs) by introducing small, deliberately designed perturbations into input samples to confuse the model and force incorrect predictions after it has been trained [71].

In both attack types, adversaries typically aim to conduct either targeted or untargeted attacks [33]. In targeted attacks, the goal is to deceive the classifier into assigning a specific, predetermined label to the input sample, regardless of its actual class. In contrast, untargeted attacks aim solely to mislead the classifier into making an incorrect prediction, without regard to which incorrect label is assigned. Targeted and untargeted attacks can be viewed similarly in the context of malware detection, as the classifiers used are typically binary.

Moreover, recent research highlights the phenomenon of *transferability*, wherein AEs or poisoning points crafted for one model can also compromise others, even if they differ in architecture or training data [72]. This property significantly broadens the attack surface, enabling adversaries to mount effective attacks even without direct access to the target model.

Generating AEs in the malware domain is more challenging compared to other domains, primarily because AEs must remain functional malicious software. Indeed, unlike some fields (e.g., computer vision), where AEs only need to deceive a model, adversarial malware must also retain its ability to compromise victim machines while evading detection. Figure 2.1 illustrates the pipeline of malware detection, highlighting the process of constructing and utilizing a malware classifier. One of the key concepts demonstrated in the figure is the distinction between the problem space and the feature space. Unlike domains such as

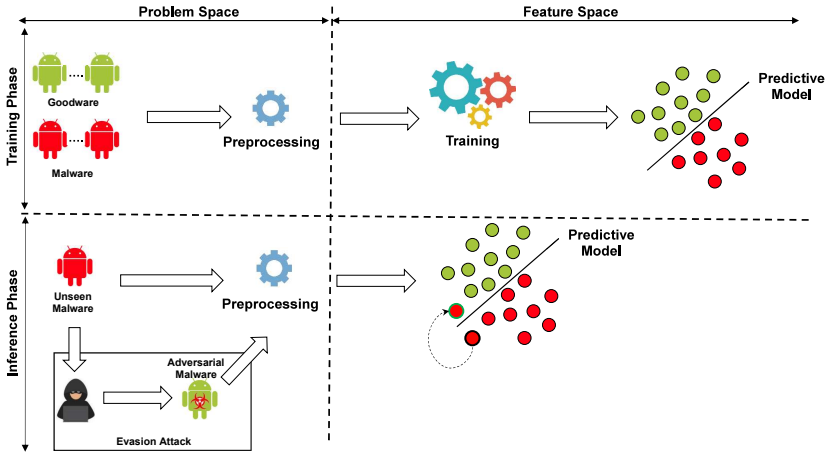


Figure 2.1: The pipeline of ML-based malware detection and its susceptibility to evasion attacks. Green and red dots represent benign and malicious samples, respectively.

computer vision, where the problem space (e.g., images) and the feature space (e.g., pixel representations) are closely aligned, in malware detection, these two spaces are significantly different. The problem space consists of real-world software (e.g., executable files), while the feature space represents the abstract representations derived from these files, which are used for ML. This separation presents unique challenges in the malware domain, particularly when generating adversarial malware capable of evading detection while preserving its functionality. Since adversarial methods typically operate in the feature space [73], there arises a need to reconstruct problem-space objects from their feature representations, adding an extra layer of complexity to the process.

2.2.1 Machine Learning Vulnerabilities to Evasion Attacks

As illustrated in Figure 2.1, evasion attacks occur during the inference phase by adversarially manipulating input malware. To mitigate their potentially catastrophic impact, it is crucial to understand the underlying reasons for the vulnerability of ML systems to such adversarial attacks[1]. Generally, as depicted in Figure 2.2, two primary factors contribute to this vulnerability: *learning vulnerability* and *feature vulnerability* [8].

- Learning vulnerability. Figure 2.2 illustrates how the decision boundary learned during the training process may differ from the truly optimal decision boundary. This discrepancy creates *blind spots*, representing areas where malicious samples are misclassified as benign. These blind spots emerge due to the gap between the optimal decision boundary and the learned decision boundary, often caused by limitations in the training process. Learning vulnerability may stem from the statistical and probabilistic nature of ML, where inherent discrepancies between the ground truth distribution and the finite training samples introduce error, making it difficult to fully approximate

[1] Since the thesis specifically focuses on evasion attacks, the term *adversarial attacks* will always refer to evasion attacks throughout the rest of the thesis.

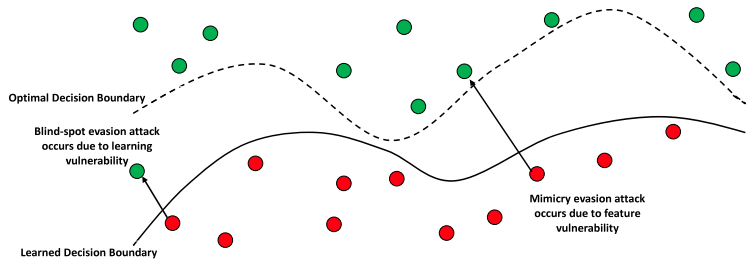


Figure 2.2: Illustrating the learning and feature vulnerabilities of malware classifiers, which are exploited by blind-spot and mimicry evasion attacks, respectively. Green and red samples represent benign and malicious samples, respectively.

the optimal decision boundary. Achieving adversarial robustness typically seems to require significantly more data than standard training, highlighting the limitations of learning from finite datasets [74]. Moreover, AEs can emerge due to test-time errors under noisy conditions, reflecting the uncertainty inherent in model behavior [75]. Blind-spot evasion attacks exploit vulnerabilities arising from these limitations by manipulating malicious samples to fall into regions where the model misclassifies them as benign.

- Feature vulnerability. As shown in Figure 2.2, adversaries can manipulate the feature representations of malicious samples to resemble those of benign ones. This manipulation becomes possible when the feature representation used to map objects from the problem space to feature vectors in the feature space lacks sufficient expressiveness to effectively distinguish between malware and goodware. In such cases, different inputs from the problem space—such as benign and malicious applications—may be mapped to similar or indistinguishable feature vectors, undermining the classifier's ability to separate them effectively. A poorly designed or insufficiently discriminative feature representation increases the risk of mimicry evasion attacks bypassing the classifier by exploiting these representational limitations.

While classification errors derived from learning vulnerability are reducible through methods such as adversarial training, which involves training the classifier on AEs to enhance its detection capabilities, classification errors due to feature vulnerability constitute a non-reducible error that cannot be resolved merely by modifying the learning algorithm [8].

2.2.2 Threat Models of Evasion Attacks

When attempting to bypass malware classifiers, adversaries rely on various threat models, which are defined by the following critical attributes:

Adversarial Knowledge. Evasion attacks are categorized based on the adversary's understanding of the target malware classifiers, including details such as training data, feature representations, learning algorithms, and the trained models. These levels of knowledge determine the following attack's complexity:

- Perfect Knowledge (PK): In this scenario, adversaries have complete knowledge of the target malware detection system, including its architecture, parameters, and training details. Such attacks are referred to as white-box attacks because they allow adversaries to exploit every aspect of the classifier to generate adversarial malware.

- Zero Knowledge (ZK): Here, adversaries lack any information about the target system. These attacks referred to as black-box attacks, rely on techniques like query-based exploration [76] or surrogate models [77] to bypass the target malware classifiers without requiring access to their internal details.

- Limited Knowledge (LK): Adversaries have partial information about the malware classifier, such as the knowledge of the feature representation but not the exact training data or model parameters. Such attacks are referred to as gray-box attacks as they balance between black-box and white-box scenarios.

Adversarial Capabilities. Using their available knowledge, adversaries manipulate either the feature space or the problem space to craft adversarial malware. These manipulations aim to force the malware classifier into misclassifying malicious samples as benign.

- Feature-Space Perturbations: Adversaries identify and apply small, carefully crafted modifications to the feature representations of malicious software. While this approach is straightforward within the feature space, these perturbations must be mapped back to the problem space to create functional malware.

- Problem-Space Manipulations: Adversaries manipulate the malware's code or binary files directly to create adversarial malware in the problem space, ensuring it remains realizable by adhering to domain constraints in the problem space, including available transformations, preserved semantics, robustness to preprocessing, and plausibility [9]. Specifically, adversaries require appropriate problem-space transformations to manipulate malware in the problem space. When applying these transformations, the malware must preserve its original malicious functionality (e.g., stealing data or exploiting vulnerabilities) while avoiding detection during feature extraction or preprocessing. Additionally, the modified malware must appear valid to bypass malware classifiers, maintaining its plausibility as benign software.

Table 2.1 provides an overview of key aspects—such as attacker assumptions and manipulation techniques—considered in existing evasion attacks, highlighting the diversity of these aspects across the literature.

2.2.3 Defenses Against Evasion Attacks

To enhance the resilience of malware classifiers against evasion attacks, we must improve their robustness against adversarial manipulations. Robustness, in general, refers to the ability of a system to maintain consistent performance despite uncertainties or variations in input parameters [78]. In ML systems, such as malware classifiers, a robust classifier should accurately classify inputs even when adversaries deliberately modify them to evade detection. Figure 2.3 conceptually illustrates the difference between a robust and a non-robust malware classifier. A malware classifier is considered robust within a perturbation range ϵ,

Table 2.1: Overview of key aspects of evasion attacks in malware detection.

Aspects	Findings
Attack Strategy	Majority of studies employ gradient-free strategies (e.g., random search [12], genetic algorithms [19, 26, 38]) or gradient-driven methods (e.g., PGD [24], JSMA [17], FGSM[a] [18]). Some explore novel designs like GANs[b] [11] or reinforcement learning [14].
Knowledge Setting	Many studies assume either white-box (e.g., [9, 17]) or gray-box (e.g., [20]) settings, where some utilize substitute models to approximate gradients [13] or query the target model to discover adversarial perturbations in the feature space of detectors [12]. In contrast, only a few studies adopt black-box settings (e.g., [26, 46]), often relying on repeated queries to target detectors.
Manipulation Space	Most studies (e.g., [14, 38]) perform perturbations directly in the feature space, while a few also consider the problem space (e.g., [9]).
Classifier Types	DNNs are the most common target models (e.g., [12, 17, 24–26, 38]), followed by SVM and other classical ML models.
Trends Over Time	Earlier studies (2017–2019) often focused on gradient-based methods, white-box assumptions, and feature-space perturbations, whereas more recent works (2020–2024) emphasize query-efficient black-box strategies and the practical realizability of attacks.

[a] Fast Gradient Sign Method
[b] Generative Adversarial Networks

known as the perturbation bound, if for every malicious input x, the classifier consistently identifies any variations (e.g., x' and x'' in Figure 2.3 (a), which are adversarially perturbed variations of x) within the ϵ-bound area as malware. Conversely, the classifier is deemed non-robust if it misclassifies such adversarial malware (e.g., x' in Figure 2.3 (b)) as goodware. This distinction highlights the critical need for robust defenses to ensure reliable malware detection in adversarial settings.

To this end, recent research has explored systematic evaluation techniques—such as explainability-guided testing frameworks—that help reveal vulnerabilities in model decision-making under adversarial conditions [79]. To address these vulnerabilities, several defense strategies have been proposed to enhance the adversarial robustness of ML systems [80]:

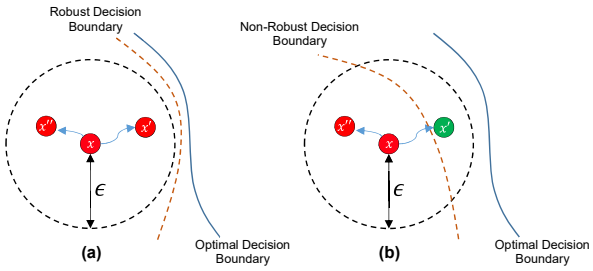


Figure 2.3: Illustration of (a) a robust and (b) a non-robust decision boundary within the perturbation bound ϵ. Green and red dots represent benign and malicious samples, respectively.

- Model robustification: This approach improves the adversarial robustness of ML systems by modifying the learning algorithms. For instance, adversarially robust optimization [53] incorporates AEs during the training phase using a min-max optimization framework to minimize the worst-case loss. Similarly, regularization [81], which adds a penalty term to the loss function to avoid overfitting, is another approach to enhance the model's resilience against adversarial perturbations.
- Input transformation: This type of defense involves transforming the input data into a new representation using techniques such as feature squeezing [82] or autoencoders [83]. The transformed data is then fed to the model, which reduces the effect of adversarial perturbations and makes it harder for attackers to bypass classifiers.
- AE detection: This defense strategy focuses on identifying adversarial inputs before they are processed by the ML system. These detection methods rely on specific criteria to distinguish valid inputs from adversarial ones, effectively filtering out potentially harmful inputs.

Throughout this dissertation, the effectiveness of these defense mechanisms is explored in the context of securing ML-based malware detection systems against evasion attacks. Model robustification using adversarially robust optimization is explored in Chapters 4 and 6, input transformation is investigated in Chapter 5, and AE detection is explored in Chapter 4. These analyses provide a comprehensive understanding of how various defense strategies can strengthen malware classifiers against adversarial threats.

3

Subverting Machine Learning in Malware Detection

Over the last decade, researchers have extensively explored the vulnerabilities of Android malware detectors to adversarial examples through the development of evasion attacks; however, the practicality of these attacks in real-world scenarios remains arguable. The majority of studies have assumed attackers know the details of the target classifiers used for malware detection, while in reality, malicious actors have limited access to the target classifiers. This chapter introduces *EvadeDroid*, a problem-space adversarial attack designed to effectively evade black-box Android malware detectors in real-world scenarios. EvadeDroid constructs a collection of problem-space transformations derived from benign donors that share opcode-level similarity with malware apps by leveraging an *n*-gram-based approach. These transformations are then used to morph malware instances into benign ones via an iterative and incremental manipulation strategy. The proposed manipulation technique is a query-efficient optimization algorithm that can find and inject optimal sequences of transformations into malware apps. Our empirical evaluations carried out on 1K malware apps, demonstrate the effectiveness of our approach in generating real-world adversarial examples in both soft- and hard-label settings. Our findings reveal that EvadeDroid can effectively deceive diverse malware detectors that utilize different features with various feature types. Specifically, EvadeDroid achieves evasion rates of 80%-95% against DREBIN, Sec-SVM, ADE-MA, MaMaDroid, and Opcode-SVM with only 1-9 queries. Furthermore, we show that the proposed problem-space adversarial attack is able to preserve its stealthiness against five popular commercial antiviruses with an average of 79% evasion rate, thus demonstrating its feasibility in the real world.

3.1 Introduction

Machine Learning (ML) continues to show promise in detecting sophisticated and zero-day malicious programs [85–91]. However, despite the effectiveness of ML-based malware detectors, these defense strategies are vulnerable to *evasion attacks* [57]. More concretely, attackers aim to deceive ML-based malware classifiers by transforming existing malware

This chapter is based on the published paper: H. Bostani and V. Moonsamy, EvadeDroid: A Practical Evasion Attack on Machine Learning for Black-Box Android Malware Detection, Computers & Security, vol. 139, 2024 [84]. The content remains unchanged from the published version.

into *adversarial examples* (AEs) via a series of manipulations. The proliferation of Android malware [92] has extended research into novel evasion attacks to strengthen malware classifiers against AEs [9, 14–23, 28, 93]. However, this endeavor, which also exists for other platforms, such as Windows, poses its own set of challenges, which we elaborate on further below.

The first challenge pertains to the *feature representation* of Android applications (apps). Making a slight modification in the feature representation of a malware app may break its functionality [57] as malware features extracted from Android Application Packages (APKs) are usually discrete (e.g., app permissions) instead of continuous (e.g., pixel intensity in a grayscale image). One plausible solution is to manipulate the features extracted from the Android Manifest file [14, 17, 21]; however, the practicality of such manipulations in generating executable AEs is questionable for the following reasons. Firstly, modifying features from the Android Manifest (e.g., content providers, intents, etc.) cannot guarantee the executability of the original apps (i.e., malicious payload) [33, 93]. Secondly, adding unused features to the Manifest file can be discarded by applying pre-processing techniques [9]. Finally, advanced Android malware detectors (e.g., [94, 95]) primarily rely on the semantics of Android apps, which are represented by the Dalvik bytecode rather than the Manifest files [21, 22].

Another challenge is the limitations of *feature mapping* techniques used to convert Android apps from the problem space (i.e., input space) to the feature space. These techniques are not reversible, meaning that feature-space perturbations cannot be directly translated into a malicious app [9]. To address *inverse feature-mapping problem*, a common approach is to manipulate real-world malware apps using problem-space transformations that correspond to the features used in ML models. By applying these feature-based transformations to Android apps, adversaries can create hazardous evasion attacks [9, 22, 23, 28]. However, finding suitable transformations that satisfy problem-space constraints is not straightforward [9]: Firstly, certain transformations (e.g., [24, 25]) intended to mimic feature-space perturbations may not result in feasible AEs because they disregard feature dependencies from real-world objects. Additionally, some transformations (e.g., [9, 23]) that meet problem-space constraints for manipulating real objects may introduce undesired or incompatible payloads into malware apps. These types of transformations not only might render the perturbations different from what the attacker expects [23] but can also lead to the crashing of adversarial malware apps.

The final challenge revolves around current methods [9, 14–25] that generate AEs based on the specifics of target malware detectors, such as the ML algorithm and feature set. These approaches assume that attackers possess either *Perfect Knowledge* (PK) or *Limited Knowledge* (LK) about the target classifiers. However, in real-world scenarios, adversaries generally have *Zero Knowledge* (ZK) about the target malware detectors, which aligns more closely with reality since antivirus systems operate as black-box engines that are queried [45]. Some studies [12, 13, 20] have explored semi-black-box settings to generate AEs by leveraging feedback from the target detectors. Nevertheless, these approaches suffer from inefficiency in terms of evasion costs, including the high number of queries required and the extent of manipulation applied to the input sample. Efficient querying is crucial due to the associated costs [45] and the risk of detectors blocking suspicious queries. Additionally, minimizing manipulation is desired as excessive manipulations could impact the malicious functionality of apps [16].

3.1.1 Contributions

In response to the challenges outlined earlier, we propose a comprehensive and generalized evasion attack called *EvadeDroid*, which can bypass black-box Android malware classifiers through a two-step process: (i) *preparation* and (ii) *manipulation*. The first step involves implementing a donor selection technique within EvadeDroid to create an action set comprising a collection of problem-space transformations, i.e., code snippets known as *gadgets*. These gadgets are derived by performing program slicing on benign apps (i.e., donors) that are publicly available. By injecting each gadget into a malware app, specific payloads from a benign donor can be incorporated into the malware app. Our proposed technique utilizes an *n-gram-based similarity* method to identify suitable donors, particularly benign apps that exhibit similarities to malware apps at the opcode level. Applying transformations derived from these donors to malware apps can enable them to appear benign or move them towards blind spots of ML classifiers. This approach aims to achieve the desired outcome of introducing transformations that not only ensure adherence to problem-space constraints (i.e., preserved semantics, robustness to preprocessing, and plausibility [9]) but also lead to malware classification errors.

In the manipulation step, EvadeDroid uses an iterative and incremental manipulation strategy to create real-world AEs. This procedure incrementally perturbs malware apps by applying a sequence of transformations gathered in the action set into malware samples over several iterations. We propose a search method to randomly choose suitable transformations and apply them to malware apps. The random search algorithm, which moves malware apps in the problem space, is guided by the labels of manipulated malware apps. These labels are specified by querying the target black-box ML classifier. Our contributions can be summarized as follows:

- We propose a *black-box evasion* attack that generates real-world Android AEs that adhere to problem-space constraints. To the best of our knowledge, EvadeDroid is the pioneer study in the Android domain that successfully evades ML-based malware detectors by directly manipulating malware samples without performing feature-space perturbations.

- We demonstrate that EvadeDroid is a *query-efficient* attack capable of deceiving various black-box ML-based malware detectors through minimal querying. Specifically, our proposed problem-space adversarial attack achieves evasion rates of 89%, 85%, 86%, 95%, and 80% against DREBIN [96], Sec-SVM [16], ADE-MA [25], MaMaDroid [95], and Opcode-SVM [97], respectively. This chapter represents one of the pioneering efforts in the Android domain, introducing a realistic problem-space attack in a ZK setting.

- Our proposed attack can operate with either *soft labels* (i.e., confidence scores) or *hard labels* (i.e., classification labels) of malware apps, as specified by the target malware classifiers, to generate AEs.

- We assess the practicality of the proposed evasion attack under real-world constraints by evaluating its performance in deceiving popular commercial antivirus products. Specifically, our findings indicate that EvadeDroid can significantly diminish the

effectiveness of five popular commercial antivirus products, achieving an average evasion rate of approximately 79%.

- In the spirit of open science and to allow reproducibility, we have made our code available at `https://github.com/HamidBostani2021/EvadeDroid`.

The rest of the chapter is organized as follows: Section 3.2 reviews the most important relevant studies, particularly in the Android domain. In Section 3.3, we provide background information on fundamental concepts, specifically ML-based malware detectors, and briefly discuss the practical transformations that can be used for manipulating APKs. Section 3.4 initiates by reviewing the threat model and articulating the problem definition for EvadeDroid. Following this, an illustration of the proposed black-box attack will be presented. We evaluate EvadeDroid's performance in Section 3.5. Limitations and future work, along with a brief conclusion, are presented in Section 5.6 and Section 3.7.

3.2 Related work

In the past few years, several studies have explored AEs in the context of malware, particularly in the Windows domain. For example, Demetrio et al. [46] generated AEs in a black-box setting by applying structural and behavioral manipulations. Song et al. [98] employed code randomization techniques to generate real-world AEs. They proposed an adversarial framework guided by reinforcement learning to model the action selection problem as a multi-armed bandit problem. Sharif et al. [47] used binary diversification techniques to evade malware detection. Khormali et al. [33] bypassed visualization-based malware detectors by applying padding and sample injection to malware samples. Demetrio et al. [99] generated adversarial malware by making small manipulations in the file headers of malware samples. Rosenberg et al. [45] presented a black-box attack that perturbs API sequences of malware samples to mislead malware classifiers.

While evasion attacks have made significant advancements in the Windows domain, their effectiveness in the Android domain may be limited because their manipulations might not be appropriate for altering Android malware apps in a way that can deceive existing Android malware detectors. Over the last few years, various studies have been performed to generate AEs in the Android ecosystem to anticipate possible evasion attacks. Table 3.1 illustrates the threat models that were considered by researchers. Note that in the categorization of studies under the ZK setting, adversaries should not only lack access to the details of the target model but also have no assumptions (e.g., types of features utilized by detectors) about it. To study feature-space AEs, Croce et al. [12] introduced Sparse-RS, a query-based attack that generated AEs using a random search strategy. Rathore et al. [14] generated AEs by using Reinforcement Learning to mislead Android malware detectors. Chen et al. [15, 18] implemented different feature-based attacks (e.g., brute-force attacks) to evaluate their defense strategies. Demontis et al. [16] presented a white-box attack to perturb feature vectors of Android malware apps regarding the most important features that impact the malware classification. Liu et al. [19] introduced an automated testing framework based on a Genetic Algorithm (GA) to strengthen ML-based malware detectors. Xu et al. [20] proposed a semi-black-box attack that perturbs features of Android apps based on the simulated annealing algorithm. The above attacks seem impractical as they do not show how real-world apps can be reconstructed based on feature-space perturbations.

Table 3.1: Evasion attacks in ML-based Android malware detectors.

Relevant Papers	Attacker's Knowledge			Perturbation Type	
	PK	LK	ZK	Problem Space	Feature Space
Xu et al. [26]			✓	✓	✓
He et al. [27]			✓	✓	✓
Li et al. [11]		✓		✓	✓
Croce et al. [12]		✓			✓
Zhang et al. [13]		✓		✓	✓
Rathore et al. [14]	✓	✓			✓
Chen et al. [15]	✓	✓			✓
Demontis et al. [16]	✓	✓	✓	✓	✓
Grosse et al. [17]	✓			✓	✓
Chen et al. [18]	✓	✓			✓
Liu et al. [19]		✓			✓
Xu et al. [20]		✓			✓
Berger et al. [21]	✓	✓		✓	✓
Pierazzi et al. [9]	✓			✓	✓
Chen et al. [22]		✓		✓	✓
Cara et al. [23]		✓		✓	✓
Yang et al. [28]			✓	✓	✓
Li et al. [24]	✓	✓		✓	✓
Li et al. [25]	✓	✓		✓	✓
EvadeDroid			✓	✓	

To investigate problem-space manipulations, Grosse et al. [17] manipulated the Android Manifest files based on the feature-space perturbations. Berger et al. [21] and Li et al. [24, 25] used a similar approach; however, they considered both Manifest files and Dalvik bytecodes of Android apps in their modification methods. Zhang et al. [13] introduced an adversarial attack called *ShadowDroid* to generate AEs using a substitute model built on permissions and API call features. Xu et al. [26] introduced GenDroid, a query-based attack that employed GA by integrating an evolutionary strategy based on Gaussian Process Regression. The practicality of these attacks is also questionable because the generated AEs might not satisfy all the constraints in the problem space [9] (e.g., plausibility and robustness to preprocessing). For instance, Li et al. [24] reported that 5 out of 10 manipulated apps that were validated could not run successfully. Furthermore, unused features injected into APKs by the attacks discussed in [13, 17, 21, 24–26] not only raise plausibility concerns but also render them susceptible to elimination by preprocessing operator [9], especially those features incorporated into Manifest files.

In addition to the aforementioned studies, some (e.g., [9, 22, 23, 28]) have considered the *inverse feature-mapping problem* when presenting practical AEs in the Android domain. Pierazzi et al. [9] proposed a problem-space adversarial attack to generate real-world AEs by applying functionality-preserving transformations to the input malware apps. Chen et al. [22] added adversarial perturbations found by a substitute ML model to Android malware apps. Cara et al. [23] presented a practical evasion attack by injecting system API calls determined via mimicry attack on APKs. Li et al. [11] proposed a problem-space attack called *BagAmmo*, targeting function call graph (FCG) based malware detection. The main shortcoming of

these studies is that the authors assume the adversary to have perfect knowledge [9] or limited knowledge [22, 23] about the target classifiers (e.g., knowing the feature space or accessing the training set), while in real scenarios (e.g., bypassing antivirus engines), an adversary often has zero knowledge about the target malware detectors. For instance, BagAmmo [11] assumes that the target malware detector is based on FCG, which implies that it has some knowledge about the target model. Note that this assumption may not be applicable in all real-world scenarios, as different malware detectors may employ diverse feature sets.

On the other hand, despite the practicality of [9] in attacking white-box malware classifiers, the side-effect features that appear from undesired payloads injected into malware samples may manipulate the feature representations of apps differently from what the attacker expects [23]. Furthermore, such attacks may cause the adversarial malware to grow infinitely in size as they do not consider the size constraint of the adversarial manipulations. The attacks presented in [11, 22] are tailored to the target malware classifiers (i.e., DREBIN [96], and FCG-based detectors such as MaMaDroid [95]), which means the authors did not succeed in presenting a generalized evasion technique. Moreover, the attack in [23] has some limitations, such as injecting incompatible APIs into Android apps or using incorrect parameters for API calls, which can crash adversarial malware apps.

To address the aforementioned shortcomings, Yang et al. [28] proposed two attacks named the *evolution* and *confusion* attacks, designed to evade target classifiers in a black-box setting. However, their approach lacks details about critical issues (e.g., the feature extraction method) and is impractical because, as reported by the authors, their attacks can easily disrupt the functionality of APKs after a few manipulations. Demontis et al. [16] employed an obfuscation tool to bypass Android malware classifiers, but their results indicate a low performance for their method. He et al. [27] introduced a query-based attack utilizing a perturbation selection tree and an adjustment policy. Nevertheless, the proposed attack is ineffective in hard-label settings, which are crucial for most real-world scenarios. Furthermore, in addition to the questionable plausibility of this attack, its success would be jeopardized by the disputable assumption that perturbations in the attack's malware perturbation set impact the feature values of target malware detectors.

EvadeDroid addresses the limitations of existing attacks by thoroughly aiming to meet the practical demands of real-world scenarios, such as hard-label attacking in a fully ZK setting, query efficiency, and satisfaction of all problem-space constraints. The novelty of our work, compared to the aforementioned studies, lies in the following aspects: (i) EvadeDroid provides adversaries with a general tool to bypass various Android malware detectors, as it is a problem-space evasion attack that operates in a ZK setting without any pre-assumptions about the features and types of features employed by the target malware detectors (Section 3.5.2). (ii) Unlike other evasion attacks, EvadeDroid directly manipulates Android apps without relying on feature-space perturbations. Its transformations not only are independent of the feature space but also adhere to problem-space constraints (Section 3.4.1). (iii) EvadeDroid is simple and easy to implement in real-world scenarios (Section 3.5.4) with proper transferability (Section 3.5.5). It is a query-efficient evasion attack that only requires the hard labels of Android apps provided by target black-box malware detectors (e.g., cloud-based antivirus services) (Section 3.5.2).

3.3 Background

In this section, we present a concise overview of of the fundamental backgrounds relevant to Android evasion attacks. This encompasses the structure of Android apps, ML-based Android malware detection, and the adversarial transformations used for generating Android adversarial malware.

3.3.1 Android Application Package (APK)

APK is a compressed file format with a *.apk* extension. APKs contain various contents such as Resources and Assets. However, the most crucial contents, particularly for malware detectors, are the Manifest (AndroidManifest.xml) and Dalvik bytecode (classes.dex). The Manifest is an XML file that provides essential information about Android apps, including the package name, permissions, and definitions of Android components. It contains all the metadata required by the Android OS to install and run Android apps. On the other hand, Dalvik bytecode, also known as Dalvik Executable or DEX file, is an executable file that represents the behavior of Android apps.

Apktool [100] is a popular reverse-engineering tool for the static analysis of Android apps. This reverse-engineering instrument can decompile and recompile Android apps. In the decompilation process, the DEX files of Android apps are decompiled into a human-readable code called *smali*. Besides the above tool, *Soot* [101] and *FlowDroid* [102] are two Java-based frameworks that are used for analyzing Android apps. Soot extracts different information from APKs (e.g., API calls) which are then used during static analysis. One of the advantages of Soot for malware detection is its ability to generate call graphs; however, Soot cannot generate accurate call graphs for all apps because of the complexity of the control flow of some APKs. To address this shortcoming, FlowDroid, which is a Soot-based framework, can create precise call graphs based on the app's life cycle. It is worth noting that EvadeDroid uses Apktool, FlowDroid, and Soot in different components of its pipeline to generate adversarial examples.

3.3.2 ML-Based Android Malware Detection

Leveraging ML for malware detection has garnered significant interest among cybersecurity researchers in the past decade. ML has demonstrated its potential as an effective solution in static malware analysis, enabling the identification of sophisticated and previously unknown malware through the generalization capabilities of ML algorithms [57]. It is important to note that static analysis is a prominent approach for detecting malicious programs, where apps are classified based on their source code (i.e., static features) without execution. This approach offers fast analysis, allowing for the examination of an app's code comprehensively, with minimal resource usage in terms of memory and CPU [103]. In order to represent programs for ML algorithms, various types of features are commonly employed in the static analysis, including syntax features (e.g., requested permissions and API calls [16, 25, 96]), opcode features (e.g., n-gram opcodes [104]), image features (e.g., grayscale representations of bytecodes [105]), and semantic features (e.g., function call graphs [95]).

3.3.3 Adversarial Transformations

In the programming domain, a safe transformation refers to a problem-space transformation that maintains the semantic equivalence of the original program while ensuring its excitability. In the adversarial malware domain, safe transformations, which guarantee preserved-semantics constraint, can become adversarial transformations if they are also plausible and robust to processing (refer to Appendix 3.A for additional details regarding these constraints). Generally, in the context of Android malware detection, attackers have three types of adversarial transformations at their disposal to manipulate malicious apps [9]: (i) *feature addition*, (ii) *feature removal*, and (iii) *feature modification*. Feature addition involves adding new elements, such as API calls, to the programs, while feature removal entails removing contents like user permissions. Feature modification combines both addition and removal transformations in malware programs. Most studies have primarily focused on feature addition, as removing features from the source code is a complex operation that may cause malware apps to crash. Code transplantation [9, 28], system-predefined transformation [23], and dummy transformation [17, 21, 22, 24, 25] are three potential methods for adding features to manipulate Android apps. However, two main issues arise when considering feature additions:

(i) What specific content should be included. By deriving problem-space transformations from feature-space perturbations, the attacker aims to ensure that the additional contents (e.g., API calls, Activities, etc.) are guaranteed to appear in the feature vector of the manipulated malware app [9]. Therefore, attackers may either use dummy contents (e.g., functions, classes, etc.) [22] or system-predefined contents (e.g., Android system packages) [23] for this purpose. As the plausibility of these transformations is debatable due to the potential lack of complete inconspicuousness, malicious actors may also make use of content present in already-existing Android apps. The *automated software transplantation* technique [106] can then be used to allow attackers to successfully carry out safe transformations. They extract some slices of existing bytecodes from benign apps (i.e., donor) during the *organ harvesting* phase, and the collected payloads are injected into malware apps in the *organ transplantation* phase.

(ii) Where contents should be injected. New contents must preserve the semantics of malware samples; therefore, they should be injected into areas that cannot be executed during runtime. For example, new contents can be added after `RETURN` instructions [16] or inside an `IF` statement that is always false [9]. However, these injected contents are not robust to preprocessing if static analysis can discard unreachable code. One creative idea to add unreachable code that is undetectable is the use of *opaque predicates* [107]. In this approach, new contents are injected inside an `IF` statement where its outcome can only be determined at runtime [9].

3.4 Proposed Attack

Here we first review the threat model and the problem definition of EvadeDroid. Subsequently, we will offer an illustration of the proposed attack.

3.4.1 Threat Model

Adversarial Goal. The purpose of EvadeDroid is to manipulate Android malware samples in order to deceive static ML-based Android malware detectors. The proposed attack is an untargeted attack [108] designed to mislead binary classifiers utilized in Android malware detection, causing Android malware apps to be misclassified. In other words, EvadeDroid's objective is to trick malware classifiers into classifying malware samples as benign.

Adversarial Knowledge. The proposed evasion attack has black-box access to the target malware classifier. Therefore, EvadeDroid does not have knowledge of the training data D, the feature set X, or the classification model f (i.e., the classification algorithm and its hyperparameters). The attacker can only obtain the classification results (e.g., hard labels or soft labels) by querying the target malware classifier.

Adversarial Capabilities. EvadeDroid is designed to deceive black-box Android malware classifiers during their prediction phase. Our attack manipulates an Android malware app by applying a set of safe transformations, known as Android gadgets (i.e., slices of the benign apps' bytecode), which are optimized through interactions with the black-box target classifier. To ensure adherence to problem-space constraints, EvadeDroid leverages a previous tool [9], for extracting and injecting gadgets. Furthermore, in order to avoid major disruptions to apps, the manipulation process of a malware app is conducted gradually, making it resemble benign apps. This is achieved by injecting a minimal number of gadgets extracted from benign apps into the malware app, and the process continues until the malware app is misclassified or reaches the predefined evasion cost. In addition to the problem-space constraints discussed in previous research [9], EvadeDroid must also adhere to two additional constraints highlighting the significance of minimizing evasion costs:

- **N**umber of queries. EvadeDroid is a decision-based adversarial attack that aims to generate AEs while minimizing the number of queries, thus reducing the associated costs [45].
- **Size of** adversarial payloads. In order to generate executable and visually inconspicuous AEs, such as those with minimal file size [46], EvadeDroid aims to minimize the size of injected adversarial payloads.

It is worth mentioning, each gadget consists of an organ, which represents a slice of program functionality, an entry point to the organ, and a vein, which represents an execution path that leads to the entry point [9]. EvadeDroid extracts gadgets from benign apps by identifying entry points, which are typically API calls, through string analysis. The proposed attack assumes that the benign apps used for gadget extraction are not obfuscated, particularly in terms of their API calls. This is because EvadeDroid relies on string analysis to identify entry points, which limits its ability to extract gadgets from obfuscated apps. The gadget injection is considered successful when both the classification loss value of the manipulated app increases and the injected adversarial payload conforms to the predefined size of the adversarial payload. Additionally, the injected gadgets are placed within the block of an obfuscated condition statement that is always evaluated as `False` during runtime and cannot be resolved during design time.

Defender's Capabilities. The study conducted in this chapter assumes that the target ML models do not employ adaptive defenses that are aware of the operations performed by EvadeDroid due to disclosing detectors' vulnerability to EvadeDroid. Specifically, these

target models are unable to enhance their resilience by incorporating AEs generated by EvadeDroid during adversarial training. Furthermore, they lack the capability to detect and block queries from EvadeDroid if they become suspicious of its origin. Importantly, our analysis suggests that EvadeDroid can still be effective even if we relax the second assumption regarding the defender's capabilities. This is supported by empirical evidence demonstrating that our attack often requires only a minimal number of queries to generate AEs.

3.4.2 Problem Definition

Suppose $\phi : Z \to X \subset \mathbb{R}^n$ is a feature mapping that encodes an input object $z \in Z$ to a feature vector $x \in X$ with dimension n. We denote this as $\phi(Z) = X$. Here, Z represents the input space of Android applications, and X represents the feature space of the app's feature vectors. Furthermore, let $f : X \to \mathbb{R}^2$ and $g : X \times Y \to \mathbb{R}$ denote a malware classifier and its discriminant function, respectively. The function f assigns an Android app $z \in Z$ to a class $f(\phi(z)) = \arg\max_{y=0,1} g_y(\phi(z))$, where $y = 1$ indicates that z is a malware sample and vice versa. The confidence score (soft label) for classifying z into class y is denoted as $g_y(\phi(z))$. Let $T : Z \xrightarrow{\delta \subseteq \Delta} Z$ be a transformation function, denoted as $T_{\delta \subseteq \Delta}(z) = z'$ or simply $T_\delta(z) = z'$, which transforms $z \in Z$ to $z' \in Z$ by applying a sequence of transformations $\delta \subseteq \Delta$ such that z and z' have the same functionality. Here, $\Delta = \{\delta_1, \delta_2, ..., \delta_n\}$ represents an action set consisting of safe manipulations (transformations). Each $\delta_i \in \Delta$ can independently preserve the functionality of a malware sample when applied.

The objective of the proposed evasion attack in this chapter is to generate an adversarial example $z^* \in Z$ for a given malware app $z \in Z$ by applying a minimal sequence of transformations $\delta \subseteq \Delta$ to the app, using at most Q queries, while ensuring that the amount of injected adversarial payloads is equal to or lower than α. This can be formulated as the following optimization problem:

$$\begin{aligned} \min_{\delta \subseteq \Delta} \quad & |\delta| \\ \text{s.t.} \quad & f(\phi(T_\delta(z))) \neq f(\phi(z)) \\ & q \leq Q \\ & c(T_\delta(z), z) \leq \alpha \end{aligned} \tag{3.1}$$

where $|\delta|$ denotes the cardinality of δ. Additionally, Q and α represent the evasion cost constraints of EvadeDroid, indicating the maximum query budget and the maximum size of adversarial payloads, respectively. The size of adversarial payloads refers to the relative increase in the size of a malware sample after applying δ, and it is measured using the following payload-size cost function:

$$c(T_\delta(z), z) = \frac{[T_\delta(z)] - [z]}{[z]} \times 100 \tag{3.2}$$

where $[.]$ represents the size of an APK. Equation (3.1) can be translated into the following optimization problem to find an optimal subset of transformations in the action set:

$$\begin{aligned} \underset{\delta \subseteq \Delta}{\arg\max} \quad & g_{y=0}(\phi(T_\delta(z))) \\ \text{s.t.} \quad & q \leq Q \\ & c(T_\delta(z), z) \leq \alpha \end{aligned} \tag{3.3}$$

Equation (3.3) outlines our objective to identify an optimal subset of problem-space transformations δ within the action set Δ that leads to misclassification. Specifically, the optimization aims to enhance the confidence score of classifiers in classifying $\phi(T_\delta(z))$, the feature representation of z modified by applying δ, towards the benign class indicated by 0. Note that the optimization solver is tasked with identifying the optimal δ with a maximum of Q queries, given that the adversarial payloads do not alter the size of z beyond α.

3.4.3 Methodology

The primary goal of EvadeDroid is to transform a malware app into an adversarial app in such a way that it retains its malicious behavior but is no longer classified as malware by ML-based malware detectors. This is achieved through an iterative and incremental algorithm employed in the proposed attack, which aims to disguise malware APKs as benign ones. The attack algorithm generates real-world AEs from malware apps using *problem-space transformations* that satisfy problem-space constraints. These transformations are extracted from benign apps in the wild, which are similar to malware apps using an *n-gram*-based similarity. In this approach, a *random search* algorithm is used to optimize the manipulations of apps. Each malware app undergoes incremental manipulation during the optimization process, where a sequence of transformations is applied in different iterations. Before delving into the details of the methodology, we offer a brief overview of n-grams and random search.

***n*-Grams** are contiguous overlapping sub-strings of items (e.g., letters or opcodes) with a length of n from the given samples (e.g., texts or programs). This technique captures the frequencies or existence of a unique sequence of items with a length of n in a given sample. In the area of malware detection, several studies have used n-grams to extract features from malware samples [109–113]. These features can be either byte sequences extracted from binary content or opcodes extracted from source codes. n-Grams opcode analysis is one of the static analysis approaches for detecting Android malware that has been investigated in various related works [97, 114–117]. To conduct such an analysis, the DEX file of an APK is disassembled into smali files. Each smali file corresponds to a specific class in the source code of the APK that contains variables, functions, etc. n-Grams are extracted from the opcode sequences that appear in different functions of the smali files.

Random Search (RS) [118] is a simple yet highly exploratory search strategy that is used in some optimization problems to find an optimal solution. It relies entirely on randomness, which means RS does not require any assumptions about the details of the objective function or transfer knowledge (e.g., the last obtained solution) from one iteration to another. In the general RS algorithm, the sampling distribution S and the initial candidate solution $x^{(0)}$ are defined based on the feasible solutions of the optimization problem. Then, in each iteration t, a solution $x^{(t)}$ is randomly generated from S and evaluated using an objective function regarding $x^{(t-1)}$. This process continues through different iterations until the best solution is found or the termination conditions are met. It's noteworthy that RS can be a search strategy with high query efficiency in generating AEs [12].

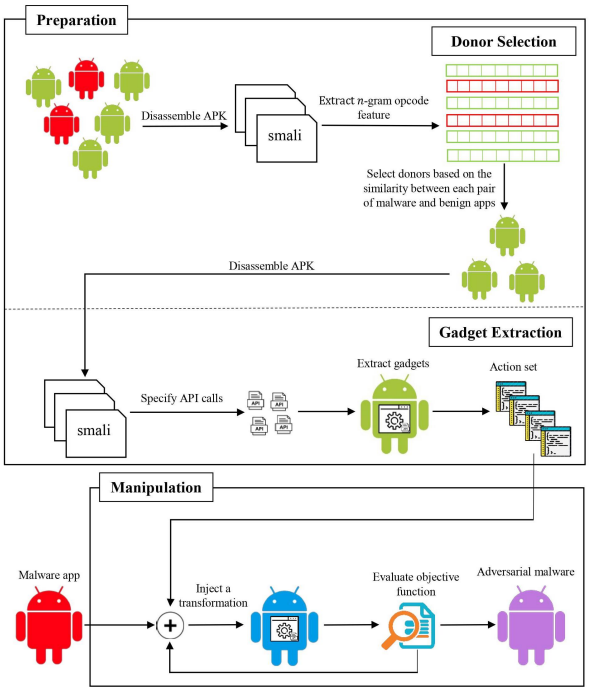


Figure 3.1: Overview of EvadeDroid's pipeline.

The workflow of the attack pipeline is illustrated in Figure 3.1, which consists of two phases: (i) preparation and (ii) manipulation.

3.4.3.1 Preparation

The primary objective of this step is to construct an action set comprising a collection of safe transformations that can directly manipulate Android applications. Each transformation in the action set should be capable of altering APKs without causing crashes while preserving their functionality. Program slicing [119], implemented in [9], is utilized in this chapter to extract the gadgets that make up the transformations collected in the action set. During the preparation step, two important considerations are determining appropriate donors and identifying suitable gadgets. Employing effective gadgets enables the modification of a set of features that can alter the classifier's decision. EvadeDroid achieves this by executing the following two sequential steps:

a) Donor selection. EvadeDroid selects donors from a pool of benign apps in order to mimic malware instances as benign ones. While it is possible to extract gadgets from any available benign app, collecting transformations from a large corpus of apps is computationally expensive due to the complexity of the program-slicing technique used for organ harvesting. Additionally, identifying potential donors resembling malware apps can lead to obtaining transformations that facilitate disguising malware apps as benign. This is because malware

apps that share similarities with benign ones may require fewer transformations to become AEs. In this chapter, EvadeDroid adopts a strategy of limiting the number of donors, i.e., choosing donors from the pool of benign apps that resemble malware apps. Our empirical results demonstrate that utilizing transformations from such benign apps accelerates the process of converting malware apps into benign ones, resulting in a reduced number of queries and transformations required for manipulation (refer to Appendix 3.B for more details).

More specifically, by utilizing the extracted gadgets from these donors, EvadeDroid can generate effective adversarial perturbations by considering both feature and learning vulnerabilities [8, 120]. Figure 3.2 provides a conceptual representation of EvadeDroid's performance in evading the target classifier. As depicted in Fig. 3.2, incorporating segments of benign apps that resemble malware apps can either make malware apps look benign ($T_\delta(z) = z_1^*$ where $\delta = \{\delta_1, \delta_2, \delta_3\}$), or shift them towards the blind spots of the target classifier (e.g., $T_\delta(z) = z_2^*$ where $\delta = \{\delta_4, \delta_5\}$). Note that some sequences of transformations may fail to generate successful AEs (e.g., $\{\delta_6, \delta_7\}$). In this work, we employ an n-gram-based opcode technique to assess the similarities between malware and benign samples. Extracting n-gram opcode features enables automated feature extraction from raw bytecodes, allowing EvadeDroid to measure the similarity between real objects without requiring knowledge of the feature vector of Android apps in the feature space of the target black-box malware classifiers. We extract n-grams following typical approaches found in the literature (e.g., [104, 121]), but with a focus on opcode types rather than the opcodes themselves. The n-gram opcode feature extraction utilized in the study conducted this chapter involves the following main steps:

1. Disassemble Android application's DEX files into smali files using Apktool.
2. Discard operands and extract n-grams from the types of all opcode sequences in each smali file belonging to the app. For example, consider a sequence of opcodes in a smali file: *I: if-eq M: move G: goto I: if-ne M: move-exception G: goto/16 M: move-result*. In this case, we have 7 opcodes with 3 types (i.e., I, M, G). Note *IM, MG, GI, GM* are all unique 2-grams that appeared in the given sequence.
3. Map the extracted feature sets to a feature space H by aggregating all observable n-grams from all APKs.
4. Create a feature vector $h \in H$ for each app, where each element of h indicates the presence or absence of a specific n-gram in the app.

Suppose M and B represent the sets of malware and benign apps, respectively, available to EvadeDroid. The similarity between each pair of a malware app $m_i \in M$ and a benign app $b_j \in B$ is determined by measuring the containment [104, 121] of b_j in m_i using the following approach:

$$\sigma(m_i, b_j) = \frac{|v(m_i) \cap v(b_j)|}{|v(b_j)|} \tag{3.4}$$

where $v(m_i)$ and $v(b_j)$ represent the sets of features with values of 1 in h_{m_i} and h_{b_j}, respectively, and $|.|$ denotes the number of features. Specifically, $|v(m_i) \cap v(b_j)|$ denotes the number of common features between m_i and b_j. It is worth emphasizing that most Android

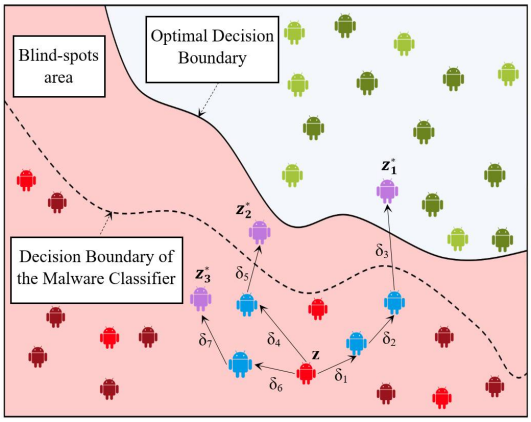


Figure 3.2: The functionality of EvadeDroid in generating real-world adversarial malware apps. The dark red and dark green samples are, respectively, the inaccessible malware and benign samples that have been used for training the malware classifier. Light red and light green samples represent, respectively, accessible malware and benign samples in the wild. The blue and purple samples are manipulated malware apps and AEs, respectively.

malware apps are created using repackaging techniques, where attackers disguise malicious payloads in legitimate apps [122]. Therefore, we consider the containment of benign samples in malware samples to determine the similarities between each pair of malware and benign samples. To identify suitable donors, we calculate a weight for each benign app $b_i \in B$ according to equation (3.4):

$$w_{b_j} = \frac{\sum_{\forall m_i \in M} \sigma(m_i, b_j)}{|M|} \tag{3.5}$$

where $|M|$ represents the number of malware apps. We then sort the benign apps in descending order based on their corresponding weights. Finally, we select the top-k benign apps as suitable donors for gadget extraction. Note that w_{b_j} reflects how closely b_j aligns with the distribution of malware apps, offering a measure of its resemblance to the characteristics of malware.

b) Gadget extraction. We collect gadgets based on the desired functionality we aim to extract from donors. EvadeDroid intends to simulate malware samples to benign ones from the perspective of static analysis; therefore, the payloads responsible for the key semantics of donors are proper candidates for extraction. To access the semantics of Android applications, EvadeDroid extracts the payloads containing API calls (i.e., the code snippet encompassing an API call and all its associations) since API calls represent the main semantics of apps [123, 124]. An API call is an appropriate point in the bytecode of an APK because the snippets encompassing the API calls are related to one of the app semantics. In sum, gadget extraction from donors consists of the following main steps:

1. Disassemble DEX files of donors into smali files by using Apktool.

2. Perform string analysis on each app to identify all API calls in its smali files.

Algorithm 3.1: Generating a real-world adversarial example.

Input: z: the original malware sample; Δ: the action set; L: the objective function; ϕ: the feature mapping function; c: the payload-size cost function; Q: the query budget; α: the allowed adversarial payload size.

Output: z^*: an adversarial example; δ: an optimal transformations.

```
q ← 1 ;
z* ← z;
L_best ←-∞;
δ ← Ø;
while q ≤ Q and z* is classified as a malware do
    λ ← Select a transformation randomly from Δ \δ;
    z' ← T_λ(z*);
    l = L(φ(z'));
    if c(z, z') ≤ α then
        if L_best ≤ l then
            L_best ← l;
            z* ← z';
            δ ← δ ∪ λ
        end
    end
end
return z*, δ
```

3. Extract the gadgets associated with the collected API calls from each app.

Ultimately, the action set Δ is formed by taking the union of the extracted gadgets.

3.4.3.2 Manipulation

We employ Random Search (RS) as a simple black-box optimization method to solve equation (3.3). Specifically, for each malware sample z, EvadeDroid utilizes RS to find an optimal subset of transformations δ in order to generate an adversarial example z^*. RS offers a significant advantage in terms of query reduction compared to other heuristic optimization algorithms, such as Genetic Algorithms (GAs). This is because RS only requires one query in each iteration to evaluate the current solution. Algorithm 3.1 outlines the key steps of the manipulation component in the proposed problem-space evasion attack. As depicted in Algorithm 3.1, the RS method randomly selects a transformation λ from the action set Δ to generate z^* for z. Subsequently, based on the adversarial payload size α, the algorithm applies λ to z only if it can improve the objective function L defined in equation (3.3), which corresponds to the discriminant function of the target classifier for $y = 0$.

Hard-label Setting. In Algorithm 3.1, we assume that our attack has access to the soft label of the target classifier. This means that EvadeDroid can obtain the confidence score provided by the black-box classification model when making queries. However, in real-world scenarios, such as antivirus systems, the target classifier may only provide hard labels (i.e., classification labels) for Android apps. This chapter considers two approaches, namely optimal and

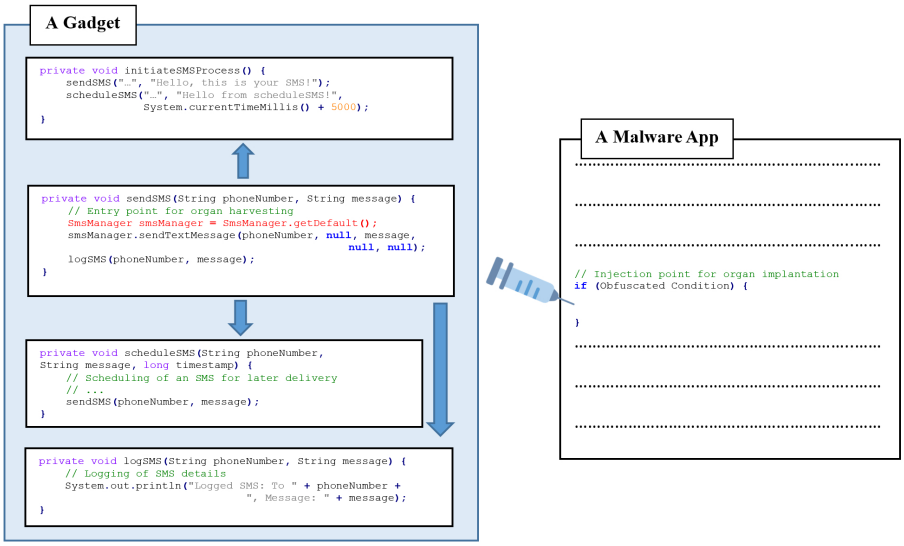


Figure 3.3: Applying a problem-space transformation (i.e., gadget) into a malware app involves injecting the gadget extracted from an API call entry point (`e.g., SmsManager`) in a donor into an obfuscated false condition statement within the malware app. The code snippets are displayed in `Java` representation to facilitate better understanding.

non-optimal hard-label attacks, to address this challenge. In the optimal hard-label attack, the adversary aims to generate AEs by applying minimal transformations. To achieve this, EvadeDroid modifies the objective function of the proposed RS algorithm (i.e., equation (3.3)) by maximizing the following objective function, while considering the evasion cost:

$$\begin{aligned} \underset{\delta \subseteq \Delta}{\arg\max} \quad & s(T_\delta(z)) \\ \text{s.t.} \quad & q \leq Q \\ & c(T_\delta(z), z) \leq \alpha \end{aligned} \tag{3.6}$$

where Q (i.e., number of queries) and α (i.e., size of adversarial payloads) represent evasion cost budgets, and c denotes the payload-size cost function (equation (3.2)). Moreover, s is the following similarity function:

$$s(a) = \max_{\forall b \in B} \frac{|v(a) \cap v(b)|}{\|h_a - h_b\|_1} \tag{3.7}$$

where B represents all available benign apps in the wild. $v(a)$ and $v(b)$ represent the sets of features with values of 1 in h_a (i.e., the feature vector of a) and h_b (i.e., the feature vector of b), respectively, and $|.|$ denotes the number of features. Furthermore, $\|h_a - h_b\|_1$ denotes the sum of the absolute differences (i.e., l_1-norm) between the opcode-based feature vectors of a and b. The l_1-norm enhances the accuracy of our similarity measurement, particularly in scenarios where the number of common features between various pairs of malware samples and benign samples is the same, aiding EvadeDroid in identifying the maximum similarity. Note that equation (3.7) aims to measure the similarity between two apps based on not only

a large set of common features but also a small distance. The underlying idea behind the introduced objective function is rooted in our primary approach to misleading malware classifiers. In other words, a transformation can be applied to a malware app if it maintains or increases the maximum similarity between the malware app and available benign apps.

On the other hand, in the non-optimal hard-label attack, EvadeDroid applies random transformations to malware until it creates an AEs or reaches the predefined query budget. Specifically, in this setting, EvadeDroid randomly selects and applies a transformation from the action set to the malware app in each query. The target classifier is then queried to determine the label of the modified app. If the label indicates that the app is still classified as malware, EvadeDroid repeats this process.

It is important to highlight that Figure 3.3 depicts the procedure of manipulating an Android malware app through a problem-space transformation, specifically injecting an extracted gadget into a malware app. For more detailed information on the implementation of EvadeDroid, we refer the reader to 3.C.

3.5 Simulation Results

In this section, we empirically assess the performance of EvadeDroid in deceiving various academic and commercial malware classifiers. Our experiments aim to answer the following research questions:

RQ1. How does the evasion cost affect the performance of EvadeDroid? (Section 3.5.2)
RQ2. Is EvadeDroid a versatile attack that can evade different Android malware detectors without relying on any specific assumptions? (Section 3.5.2)
RQ3. How does the performance of EvadeDroid compare to other similar attacks? (Section 3.5.3)
RQ4. Is EvadeDroid applicable in real-world scenarios? (Section 3.5.4)
RQ5. How does EvadeDroid demonstrate its performance despite the restriction of not being able to query the target detectors? (Section 3.5.5)
RQ6. How does the proposed RS-based manipulation strategy affect the performance of EvadeDroid? (Section 3.5.6)

All experiments have been run on a Debian Linux workstation with an Intel (R) Core (TM) i7-4770K, CPU 3.50 GHz, and 32 GB RAM.

3.5.1 Experimental Setup

Here, we provide an overview of the target detectors, datasets, and evaluation metrics we consider in our experiments.

3.5.1.1 Target Detectors

To ensure that our conclusions are not limited to a specific type of malware detection, we evaluate EvadeDroid against various malware detectors to demonstrate the effectiveness of the proposed attack. In particular, our evaluation focuses on assessing EvadeDroid's performance against well-known Android malware detection models, namely DREBIN [96], Sec-SVM [16], ADE-MA [25], MaMaDroid [95], and Opcode-SVM [97]. These models have been extensively studied in the context of detecting problem-space adversarial attacks in the Android domain [9, 13, 17, 22, 24]. For more details about these detectors, please refer to Appendix 3.D.

Table 3.2: Datasets used in our experiments.

Dataset	No. of Benign samples	No. of Malware Samples	Relevant Experiment
Inaccessible Dataset (Training Samples)	10K	2K	Section 3.5.2, Section 3.5.3, Section 3.5.5
	90K	10K	Section 3.5.5
Accessible Dataset (EvadeDroid's samples)	2K	1K	All

3.5.1.2 Dataset

We evaluate the performance of EvadeDroid using the dataset provided in [9]. This dataset consists of $\approx 170K$ samples, each represented using the DREBIN [96] feature set. The samples are feature representations of Android apps collected from AndroZoo [125] and labeled by [9] using a threshold-based labeling approach. These collected apps were published between January 2017 and December 2018. According to the labeling criteria in [9], an APK is considered malicious or clean if it has been detected by any 4+ or 0 VirusTotal (VT) [126] engines, respectively. It is important to note that the threshold-based labeling approach does not rely on specific engines but considers the number of engines involved [127]. Therefore, the engines used for labeling may vary from sample to sample.

Table 3.2 presents the specifications of datasets utilized in the study conducted in this chapter where their samples were randomly chosen from the collected data provided in [9]. It's worth mentioning that there is no overlap between the inaccessible and accessible datasets. EvadeDroid exclusively makes use of the accessible dataset, which comprises $2K$ benign samples for donor selection and $1K$ malware samples for the creation of AEs. To fulfill the requirement of direct utilization of apps in our problem-space attack, we collect $3K$ apps corresponding to EvadeDroid's accessible samples from AndroZoo, based on the apps' specifications provided with the dataset [9]. Our study conducted in this chapter employs two training sets with different scales (i.e., $12K$ and $100K$) for training classifiers. The proportion between benign and malware samples in the training sets is chosen to avoid spatial dataset bias [128]. Figure 3.4 illustrates the temporal distribution of the smaller training set, demonstrating the absence of temporal bias as these apps were published across various months. The larger training set follows a similar distribution. In Section 3.5.2, Section 3.5.3, and Section 3.5.5, a training set with a reasonable size (i.e., $12K$) is used due to the time-consuming preprocessing required by the apps in the MaMaDroid and Opcode-SVM, especially the former. Note that MaMaDroid and Opcode-SVM employ their own distinct feature representations, which differ from the DREBIN feature representation used in [9]. Therefore, to provide the training set for these detectors, we have to directly collect all considered apps in the training set from AndroZoo based on the specifications provided by [9]. Subsequently, the apps are embedded in the MaMaDroid and Opcode-SVM feature spaces using a feature extraction method. In the second evaluation conducted in Section 3.5.5, we employ a larger training set (incl., $100K$ samples) to train DREBIN and Sec-SVM in order to illustrate the impact of a larger training set on EvadeDroid. It is important to highlight that our empirical evaluation shows that training classifiers with more samples does not significantly alter the performance of EvadeDroid.

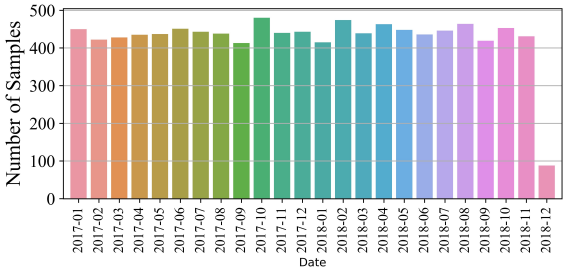


Figure 3.4: The temporal distribution of training samples. The dataset [9] lacked clarity regarding the release dates of the $\approx 1.5K$ samples in our training set.

3.5.1.3 Evaluation Metrics

We utilize the True Positive Rate (TPR) and False Positive Rate (FPR) as performance metrics for evaluating the effectiveness of malware classifiers in detecting Android malware. In Figure 3.5, we present the Receiver Operating Characteristic (ROC) curves of DREBIN, Sec-SVM, ADE-MA, MaMaDroid, and Opcode-SVM, the Android malware detectors used in this chapter, on the $12K$ training samples in the absence of our proposed attack. Note that the ROC curves were generated using 10-fold cross-validation. In addition to these metrics, we introduce the Evasion Rate (ER) and Evasion Time (ET) as EvadeDroid's performance assessment metrics in deceiving malware classifiers. ER is calculated as the ratio of correctly detected malware samples that are able to evade the target classifiers after manipulation to the total number of correctly classified malware samples. ET represents the average time, expressed in seconds, required by EvadeDroid to generate an AE, encompassing both optimization and query times. Note that the optimization time primarily consists of the execution times of random search, injecting problem-space transformations, and performing feature extraction to represent manipulated apps within the feature space. Further details of our experimental settings can be found in Appendix 3.E.

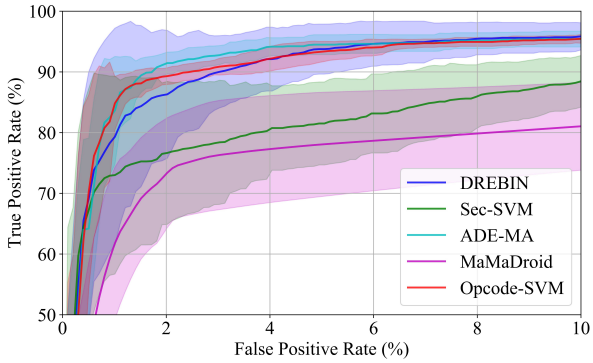


Figure 3.5: ROC curves of DREBIN, Sec-SVM, ADE-MA, MaMaDroid, and Opcode-SVM in the absence of adversarial attacks. The regions with translucent colors that encompass the lines are standard deviations.

3.5.2 Evasion Costs and Generalizability

This section first examines the influence of the allowed adversarial payload size α and the query budget Q on the performance of EvadeDroid to answer **RQ1**. Specifically, the evasion rates of EvadeDroid in fooling various malware detectors under different adversarial payload sizes and query numbers are depicted in Figure 3.6. Fig. 3.6 (a) demonstrates that the evasion rate is influenced by the size of the adversarial payload, as increasing the size allows EvadeDroid to modify more malware applications. However, we observed that for $\alpha \geq 30\%$, the impact on the evasion rate becomes less significant, as most sequences of viable transformations almost reach a plateau at $\alpha = 30\%$. Furthermore, no further improvement in evasion rates is observed beyond $\alpha = 50\%$. In addition to the adversarial payload size, the query budget is another constraint that affects the evasion rate of EvadeDroid. Fig. 3.6 (b) presents a comparison of the effect of different query numbers on the evasion rates of EvadeDroid against various malware detectors, with an allowed adversarial payload size of $\alpha = 50\%$. As can be seen in Fig. 3.6 (b), EvadeDroid requires a larger number of queries to generate successful AEs for bypassing Sec-SVM as compared to other detectors. This can be attributed to the fact that Sec-SVM, being a sparse classification model, relies on a greater number of features for malware classification compared to other classifiers. Consequently, EvadeDroid needs to apply more transformations to malware apps in order to deceive this more resilient variant of DREBIN. Additionally, Fig. 3.6 (b) demonstrates that a query budget of $Q = 20$ is nearly sufficient for EvadeDroid to achieve maximum evasion rate when attempting to bypass a malware detector. It is important to highlight that for the remaining experiments of the chapter, we have chosen to use $Q = 20$ and $\alpha = 50\%$ as they yield the optimal performance for EvadeDroid.

To answer **RQ2**, we conduct an experiment involving various malware detectors and different attack settings. Specifically, we include DREBIN, SecSVM, ADE-MA, MaMaDroid, and Opcode-SVM to cover different ML algorithms (i.e., linear vs. non-linear malware classifiers, and gradient-based vs. non-gradient-based malware classifiers) and diverse features (i.e., discrete vs. continuous features, and syntax vs. opcode vs. semantic features).

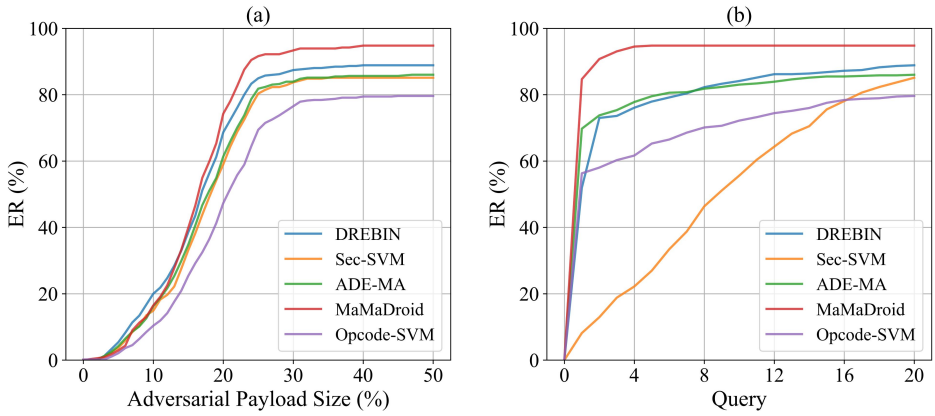


Figure 3.6: ERs of EvadeDroid operating in the soft-label setting in deceiving different Android malware detectors in terms of (a) different queries and (b) different adversarial payload sizes.

Table 3.3: Effectiveness of EvadeDroid in misleading different malware detectors when Q = 20 and α = 50%. NoQ, NoT, and AS denote Avg. No. of Queries, Avg. No. of Transformations, and Avg. Adversarial Payload Size, respectively.

Type of Threat	Target Model	ER	ET	NoQ	NoT	AS
Soft Label	DREBIN	88.9%	210.3s	3	2	15.5%
	Sec-SVM	85.1%	495.4s	9	4	16.4%
	ADE-MA	86.0%	126.2s	2	1	16.3%
	MaMaDroid	94.8%	131.4s	1	1	15.9%
	Opcode-SVM	79.6%	114.1s	3	2	18.3%
Optimal Hard Label	DREBIN	84.5%	240.6s	4	2	16.2%
	Sec-SVM	82.6%	613.1s	9	6	16.5%
	ADE-MA	84.4%	121.2s	2	1	16.3%
	MaMaDroid	94.8%	133.7s	1	1	15.9%
	Opcode-SVM	74.1%	101.2s	2	1	18.2%
Non-optimal Hard Label	DREBIN	79.7%	357.2s	4	4	16.9%
	Sec-SVM	78.2%	782.8s	9	9	17.3%
	ADE-MA	82.7%	157.3s	2	2	16.4%
	MaMaDroid	94.8%	132.6s	1	1	15.9%
	Opcode-SVM	66.6%	76.2s	1	1	18.3%

Additionally, we explore different attack settings (soft label vs. hard label) to demonstrate EvadeDroid's adaptability in various scenarios. The performance of the proposed attacks under different settings and malware detectors is presented in Table 3.3. As shown in this table, EvadeDroid demonstrates effective evasion capabilities against various malware detectors, including DREBIN, Sec-SVM, and ADE-MA with syntax binary features, as well as MaMaDroid with semantic continuous features and Opcode-SVM with opcode binary features. The evaluation also reveals that EvadeDroid performs similarly well in the optimal hard-label setting compared to the soft-label setting. It is important to note that the comparison between soft-label attacking and non-optimal hard-label attacking highlights the influence of optimizing manipulations on the performance of EvadeDroid against different detectors. While only applying transformations to malware apps is sufficient for MaMaDroid, optimizing manipulations can enhance EvadeDroid's effectiveness against other detectors, especially Opcode-SVM. For instance, our findings shown in Table 3.3 demonstrate a 13% improvement in the ER of EvadeDroid when targeting Opcode-SVM in the soft-label setting, compared to the non-optimal hard-label setting. Furthermore, when operating in the soft-label setting, EvadeDroid requires notably fewer transformations to bypass DREBIN and Sec-SVM, as compared to the non-optimal hard-label setting (e.g., 4 vs. 9 for Sec-SVM), which confirms the effectiveness of EvadeDroid in solving the optimization problem defined in equation (3.1). Table 3.3 further illustrates that our optimization leads to a substantial reduction in ET compared to the non-optimal hard-label setting. Specifically, for DREBIN and Sec-SVM, this leads to a time reduction of $\approx$ 41% and $\approx$ 37%, respectively. This significant enhancement can be attributed to the reduction in the number of transformations, achieved through the utilization of our proposed optimization technique. Note that ET brings attention to the varying time overheads associated with the feature extraction process used to compute objective values when attacking different target detectors. For example, while

NoQ and NoT are the same in attacking MaMaDroid and Opcode-SVM in the non-optimal hard-label setting, the ET for MaMaDroid is significantly higher than that for Opcode-SVM. The observed distinction is rooted in the considerable time consumption of the feature extraction process in MaMaDroid.

In summary, the results demonstrate that the proposed adversarial attack is a versatile black-box attack that does not make assumptions about target detectors, including the ML algorithms or the features used for malware detection. Furthermore, it can operate effectively in various attack settings.

3.5.3 EvadeDroid vs. Other Attacks

To answer **RQ3**, we conduct an empirical analysis to assess how EvadeDroid performs in comparison to other similar attacks. To establish a comprehensive evaluation of EvadeDroid, we consider four baseline attacks: PiAttack[1] [9], Sparse-RS [12], ShadowDroid [13], and GenDroid [26] operating in white-box, gray-box, semi-black-box, and black-box settings, respectively. These attacks serve as suitable benchmarks, allowing us to assess the performance of EvadeDroid from different perspectives, such as evasion rate and the number of queries. Similar to EvadeDroid, Sparse-RS, ShadowDroid, and GenDroid generate AEs by querying the target detectors. Additionally, PiAttack is a problem-space adversarial attack that employs a similar type of transformation to generate AEs. Although PiAttack is a white-box evasion attack, it establishes a benchmark for optimal evasion performance, facilitating the evaluation of the comparative effectiveness of other attacks with limited or zero knowledge about the targeted detectors. For further information about these attacks, please refer to Appendix 3.F. In this experiment, we chose DREBIN, Sec-SVM, and ADE-MA as the target detectors because they align with the threat models of PiAttack, Sparse-RS, and ShadowDroid. Table 3.4 shows the ERs of different adversarial attacks in deceiving various malware detectors. As can be seen in Table 3.4, although EvadeDroid has zero knowledge about DREBIN, Sec-SVM, and ADE-MA, its evasion rates for bypassing these detectors are comparable to PiAttack, where the adversary has full knowledge of the target detectors. Moreover, our empirical analysis shows that EvadeDroid requires adding more features to evade DREBIN, Sec-SVM, and ADE-MA. In concrete, on average, EvadeDroid makes 54–90 new features appear in the feature representations of the malware apps when it applies transformations to the apps for evading DREBIN, Sec-SVM, and ADE-MA, while the transformations used by PiAttack on average, trigger 11–68 features. PiAttack's ability to add a smaller number of features is attributed to its complete knowledge of the details of DREBIN, Sec-SVM, and ADE-MA. However, EvadeDroid lacks this specific information.

Furthermore, as shown in Table 3.4, the evasion rate of Sparse-RS for DREBIN and Sec-SVM demonstrates that random alterations in malware features do not necessarily result in the successful generation of AEs, even when adversaries have access to the target models' training set. Although EvadeDroid operates solely in a black-box setting, this attack outperforms Sparse-RS by a considerable margin for both DREBIN and Sec-SVM, i.e., 70.6% and 84.7% improvement, respectively. Moreover, EvadeDroid considerably surpasses ShadowDroid in attacking Sec-SVM and ADE-ME. Especially, in contrast to EvadeDroid, ShadowDroid is unsuccessful in effectively evading Sec-SVM, which is a robust detector against AEs. Note that the superior performance of ShadowDroid compared to EvadeDroid

[1] PiAttack is also referred to as the PK-Greedy attack.

Table 3.4: ERs of EvadeDroid, PiAttack, Sparse-RS, ShadowDroid, and GenDroid in misleading DREBIN, Sec-SVM, and ADE-MA. NoQ denotes Avg. No. of Queries.

Target Model	Evasion Attach	ER	NoQ
DREBIN	EvadeDroid	88.9%	3
	PiAttack	99.6%	N/A
	Sparse-RS	18.3%	195
	ShadowDroid	95.3%	31
	GenDroid	95.5%	93
Sec-SVM	EvadeDroid	85.1%	9
	PiAttack	94.3%	N/A
	Sparse-RS	0.4%	38
	ShadowDroid	8.6%	64
	GenDroid	14.5%	336
ADE-MA	EvadeDroid	86.0%	2
	PiAttack	100%	N/A
	Sparse-RS	99.7%	2
	ShadowDroid	77.8%	29
	GenDroid	100%	81

in bypassing DREBIN is based on the assumption that target detectors primarily rely on API calls and permissions. However, this assumption is not practical in real scenarios, as detectors may employ other features for malware detection. Table 3.4 further illustrates that GenDroid exhibits superior evasion rates compared to EvadeDroid when targeting DREBIN and ADE-MA; nevertheless, its efficacy is substantially nullified when facing Sec-SVM, a resilient malware detector. Our empirical analysis also highlights the remarkable efficiency of EvadeDroid in terms of the number of queries compared to other query-based attacks. Specifically, on average, EvadeDroid requires only 2–9 queries to bypass DREBIN, Sec-SVM, and ADE-MA, while Sparse-RS, ShadowDroid, and GenDroid demand 2–195, 29–64, and 81–336 queries, respectively.

In summary, the experimental results validate the practicality of EvadeDroid, which adopts a realistic threat model, in comparison to other attacks for generating AEs. Specifically, the threat models of PiAttack and Sparse-RS are essentially proposed for the detectors that operate in the DREBIN feature space, but their threat models are not practical for targeting detectors like MaMaDroid. Furthermore, ShadowDroid's effectiveness is limited to scenarios where malware detection is solely based on API calls and permissions. For instance, as demonstrated in [13], ShadowDroid is unable to deceive MaMaDroid or opcode-based detectors. In contrast, as shown in Section 3.5.2, EvadeDroid is capable of effectively fooling these types of detectors as its problem-space transformations are independent of feature space. Additionally, although GenDroid operates in the ZK setting, it perhaps encounters challenges in evading robust malware detectors like Sec-SVM and might pose potential issues in real-world scenarios due to the substantial number of queries it requires compared to EvadeDroid. Finally, Sparse-RS, ShadowDroid, and GenDroid might not be deemed realistic approaches as their abilities to satisfy problem-space constraints, particularly robustness-to-preprocessing and plausibility constraints, are questionable.

Table 3.5: Performance of EvadeDroid in the hard-label setting on five commercial antivirus products. NoM denotes No. of Detected Malware by each engine among 100 malware apps.

Engine	NoM	EvadeDroid			
		ER	Avg. Attack Time	Avg. No. of Queries	Avg. Query Time
AV1	54	68.5%	31.3s	1	214.3s
AV2	32	87.5%	54.7s	2	387.2s
AV3	31	74.2%	124.1s	2	446.6s
AV4	41	100%	35.2s	1	329.7s
AV5	11	63.6%	21.5s	1	272.9s

3.5.4 EvadeDroid in Real-World Scenarios

This experiment aims to investigate **RQ4** to demonstrate the practicality of EvadeDroid in real-world scenarios. Although the ability of EvadeDroid in the hard-label setting indicates that this attack can transfer to real life, we further consolidate this observation by measuring the impact of EvadeDroid on commercial antivirus products that are available on VT to confirm the practicality of our proposed attack in real scenarios. We chose five popular antivirus engines in the Android ecosystem based on the recent ratings of the endpoint protection platforms reported by AV-Test [129]. They are the top AVs in AV-Test capable of detecting malware apps in EvadeDroid's accessible dataset. Moreover, 100 malware apps belonging to different malware families have been randomly selected from the $1K$ malware apps available to EvadeDroid to evaluate the performance of this attack on the aforementioned five commercial detectors. To ensure the reliability of our experiment, it is crucial to confirm that the labels assigned to the malware apps used in this experiment have remained consistent. This is because the labels of collected apps are based on their corresponding samples in our benchmark dataset [9], while the labels assigned by antivirus engines to apps can potentially change over time. Therefore, we meticulously selected 100 apps that are still malware based on the threshold labeling criteria used in our primary dataset at the time of our experiment, i.e., on September 11, 2022, through querying VT. Furthermore, for each antivirus product, we generate AEs for the apps detected as malware by the antivirus. Table 3.5 presents the results of the experiment in which EvadeDroid attempts to deceive each AV in the optimal hard-label setting. In this experiment, we have assumed $Q = 10$ and $\alpha = 50\%$. As can be seen in Table 3.5, our proposed attack can effectively evade all antivirus products with a few queries. Here the effectiveness of EvadeDroid can be primarily attributed to the transformations rather than the optimization technique. This is evident from the fact that in most cases, only one query is required to generate AEs. We further investigate the performance of EvadeDroid against the overall effect of VT. Figure 3.7 shows the average number of VT detections for all 100 malware apps after each attempt of EvadeDroid to change malware apps into AEs. As depicted in Fig. 3.7, EvadeDroid can effectively deceive VT engines with an average of 70.67%. It is worth noting that the findings in this experiment validate the results observed in previous studies (e.g., [130]).

Responsible Disclosure. We conducted a responsible disclosure process to ensure the security community was informed of the findings presented in this chapter. As part of this process, we not only reached out to VT but also notified the antivirus engines that were

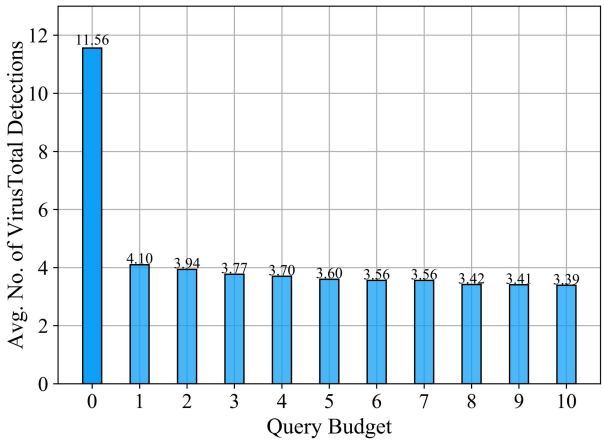


Figure 3.7: Performance of EvadeDroid in evading VT engines against different query budgets.

affected by EvadeDroid by providing detailed information about our attack methodology and sharing some test cases.

3.5.5 Transferable Adversarial Examples

In general, when decision-based adversarial attacks, such as EvadeDroid, encounter difficulty in querying specific target detectors, they can create transferable AEs using a surrogate classifier. Here we explore **RQ5** by considering transferable AEs. To investigate the transferability of EvadeDroid, we evaluate the evasion rates of AEs generated on a model (e.g., Sec-SVM), which works as a surrogate model, in misleading other target models (e.g., DREBIN). This is a stricter threat model that indicates the performance of EvadeDroid in cases where adversaries are not capable of querying the target detectors. Table 3.6 demonstrates that when EvadeDroid employs a stronger surrogate model (e.g., Sec-SVM), the AEs exhibit higher transferability. Note that the reported ERs in Table 3.6 are the evasion rates of successful AEs that are also successfully transferred.

We further compare the transferability of EvadeDroid with PiAttack [9] as it is similar to ours in terms of transformation type. This attack uses two kinds of primary features for misclassification, and side-effect features for satisfying problem-space constraints to generate realizable adversarial examples. However, EvadeDroid is not constrained by features as it operates in black-box settings. We specifically measure the transferability of the AEs in fooling Sec-SVM when DREBIN is the surrogate model. We ensure that the original apps of the AEs are correctly detected by Sec-SVM. Both DREBIN and Sec-SVM are trained with $100K$ apps (incl., $90K$ benign apps and $10K$ malware apps) to see the effect of large ML models on EvadeDroid's performance. The experimental results show that the ERs of the PiAttack and EvadeDroid in circumventing DREBIN are 99.06% and 82.12%, respectively. Furthermore, EvadeDroid is much more transferable as the transferability of the AEs generated by EvadeDroid is 58.05%, while 23.23% for PiAttack.

Table 3.6: Transferability of AEs generated by EvadeDroid.

Surrogate Model	Target Model	ER
DREBIN	Sec-SVM	25.5%
	ADE-MA	88.7%
	MaMaDroid	63.0%
	Opcode-SVM	42.2%
Sec-SVM	DREBIN	95.7%
	ADE-MA	98.5%
	MaMaDroid	95.4%
	Opcode-SVM	53.7%
ADE-MA	DREBIN	49.3%
	Sec-SVM	8.7%
	MaMaDroid	67.5%
	Opcode-SVM	22.0%
MaMaDroid	DREBIN	41.1%
	Sec-SVM	6.0%
	ADE-MA	88.9%
	Opcode-SVM	37.0%
Opcode-SVM	DREBIN	32.8%
	Sec-SVM	10.9%
	ADE-MA	66.8%
	MaMaDroid	74.83%

3.5.6 The Impact of Search Strategy on EvadeDroid

To answer **RQ6**, we perform an empirical analysis to evaluate the performance of EvadeDroid when utilizing an alternative search strategy for manipulation. Specifically, we introduce a baseline manipulation method based on GA for use in EvadeDroid, where the fitness function of the baseline is the same as the RS-based method. In the proposed GA-based manipulation method, the individuals in the population (representing potential solutions) are binary strings with a length equal to the action set Δ, where 1 indicates the corresponding transformation in Δ should be used for manipulation. This approach enhances the solution across various generations. In this experiment, the query budget for GA is set at 50 due to scalability concerns, as evaluating more solutions obtained by applying different sequences of transformations to the malware app would significantly increase time overheads. Moreover, our preliminary experiment suggests considering 9 as the population size of the GA-based method. Note that a large population size negatively affects the performance of GA, as the

Table 3.7: RS-based vs. GA-based manipulation strategies in EvadeDroid. NoQ indicates Avg. No. of Queries and NoT denotes Avg. No. of Transformations.

Search Method	ER	ET	NoQ	NoT
RS	88.9%	210.3s	3	2
GA	65.1%	630.7s	22	5

perturbation budget is quickly consumed by individuals in the initial generations.

Table 3.7 presents the results of the baseline when DREBIN is the target malware detector. As shown in Table 3.7, using RS in EvadeDroid outperforms GA. Specifically, RS not only leads to a 36.5% enhancement in ER but also accelerates EvadeDroid by $\approx 3\times$. These improvements are achieved with only 3 queries compared to GA's 22 queries.

3.5.7 Discussion

Real-world applicability. EvadeDroid demonstrates its ability to generate practical adversarial Android apps by considering real-world attack limitations, such as operating in ZK settings. We assume that EvadeDroid has no knowledge about the target malware classifiers and can only query them to obtain the labels of Android apps. Additionally, in some experiments, we assume that the target malware detectors only provide hard labels in response to the queries. The performance of EvadeDroid in various experiments validates its practicality. In a hard-label setting, it efficiently evades five popular commercial antivirus products with an average evasion rate of nearly 80%. Furthermore, empirical evaluations of EvadeDroid on DREBIN, Sec-SVM, ADE-MA, MaMaDroid, and Opcode-SVM result in evasion rates of 89%, 85%, 86%, 95%, and 80%, respectively. The success of our attack can be attributed to our approach of directly crafting adversarial apps in the problem space rather than perturbing features in the classifier's feature space. From a defender's perspective, EvadeDroid can be utilized in adversarial retraining to enhance the robustness of Android malware detection against realistic evasion attacks. Appendix 3.G includes an experiment showcasing the adversarial robustness that can be achieved with the involvement of EvadeDroid.

Functionality preserving. We extended the tool presented in [9], in particular the organ-harvesting component, to manipulate malware apps. This tool ensures the preservation of functionality by adding dead codes to malware apps without affecting their semantics. Specifically, it incorporates opaque predicates, an obfuscated condition, to inject adversarial payloads into the apps while remaining unresolved during analysis, ensuring the payloads are never executed. Generally, verifying the semantic equivalence of two programs (e.g., a malware app and its adversarial version) is not trivial [106]. Therefore, similar to the prior studies [9, 25, 28], our primary goal is to consider the installability and executability of apps to verify the correct functioning of the adversarial apps. To this end, we developed a scalable test framework that installs and executes adversarial apps on an Android Virtual Device (AVD) and conducts monkey testing [131] to simulate random user interactions with the apps to guarantee the stability of the apps. Furthermore, taking inspiration from prior research [11], we incorporate a log statement within the opaque predicate to ensure that the functionality of the manipulated apps remains unchanged. By monitoring the absence of log outputs, we can ascertain that the injected payloads are not executed. We select 50 adversarial apps, representing diverse malware families, for which their original malware apps can be installed and executed on the AVD without any issues. These apps are then subjected to our test framework. While the flaws in the Soot [101] framework (e.g., the injection of payloads through Soot might result in incorrect updates to the function address table of the app), utilized in the manipulation tool [9], affect the executability of a few cases, the majority of the apps passed the test.

Query efficiency. According to the experimental results obtained by applying EvadeDroid on academic and commercial malware detectors, we demonstrated that it can successfully carry

Table 3.8: The prevalent search strategies employed in Android evasion attacks.

Search Strategy	Description	Study
Gradient-driven	Utilizes gradients to iteratively adjust perturbations towards optimal adversarial perturbations.	[9, 13, 16–18, 22, 24, 25]
Sampling-driven	Involves exploring the solution space by sampling candidate perturbations to find optimal adversarial perturbations.	[11, 12, 14, 19, 20, 26–28]

out a query-efficient black-box attack. For instance, our proposed attack often only needs an average of 4 queries to generate the AEs that can successfully bypass DREBIN, Sec-SVM, ADE-MA, MaMaDroid, and Opcode-SVM. Moreover, we showed that EvadeDroid can effectively fool commercial antivirus products with less than two queries. One of the main reasons for being a query-efficient attack is due to the well-crafted transformations gathered in the action set. To maintain EvadeDroid's performance, it is crucial to periodically update the action set by incorporating newly published apps as new potential donors. Besides the quality of the action set, the presented optimization method is another important aspect of our proposed attack that can facilitate the identification of an optimal sequence of transformations, especially when the target detectors are robust to AEs (e.g., Sec-SVM). In fact, the proposed RS technique is an efficient sampling-driven search strategy that can quickly converge to a proper solution. Table 3.8 shows that Android evasion attacks often employ gradient-driven (e.g., gradient descent) and sampling-driven (e.g., GA) methodologies, where the latter is more practical for black-box evasion attacks because they can overcome the challenges inherent in using gradient-driven attacks in ZK settings. Specifically, gradient-driven attacks require access to precise details of target malware detectors and are limited to differentiable-based classifiers, which are not applicable to attacks operating in ZK settings. Moreover, gradient-driven techniques are not well-suited for continuous features, whereas the malware domain predominantly involves discrete features. Ultimately, gradient-masking [77] defenses implemented in target malware detectors demonstrate effectiveness in preventing gradient-driven attacks. It is important to note that our proposed sampling-driven method demonstrates greater efficiency compared to query-based methods used in other studies (e.g., [12, 20, 26, 27]). For instance, as shown in Section 3.5.3, EvadeDroid can effectively evade DREBIN with only 3 queries, whereas GenDroid needs 93 queries. Additionally, as illustrated in Section 3.5.6, our proposed RS-based strategy requires $\approx$ 210 seconds to bypass DREBIN, while the GA-based methods extend the evasion time to $\approx$ 631 seconds.

Scalability and effectiveness. Our empirical evaluations demonstrate the ability of the EvadeDroid to adapt and work effectively across a large scale of targets. Especially the results in Section 3.5.2 highlight the effectiveness of our evasion attack in bypassing diverse malware detectors (i.e., linear vs. non-linear malware classifiers, and gradient-based vs. non-gradient-based malware classifiers) that utilize different features (i.e., syntax, opcode, and semantic features) with different feature types (i.e., discrete and continuous features). Furthermore, although manipulating applications within the problem space is inherently a time-consuming

endeavor, the efficiency in querying allows our attack to autonomously generate AEs at a good speed, eliminating the need for manual and labor-intensive methods. Our empirical assessment in Section 3.5.5 also demonstrates that AEs generated by EvadeDroid to target a specific detector exhibit reusability across various malware detectors.

Potential applications. EvadeDroid shows promise for various real-world applications within the realm of Android malware detection. Security professionals and organizations involved in the development and deployment of malware detectors can utilize EvadeDroid for security testing and evaluation. For instance, they can simulate adversarial scenarios to identify vulnerabilities and enhance the robustness of their systems against real-world threats. The adversarial training capabilities of the system render EvadeDroid a helpful asset for developers seeking to strengthen malware detectors against real-world evasion attacks. Moreover, our attack can be instrumental in the development of countermeasures, allowing cybersecurity experts to understand and address potential weaknesses in existing malware detection systems.

3.6 Limitations and Future Work

In this section, we elaborate on the limitations of our proposed method, which can be considered as future work. One of the concerns of EvadeDroid is the adversarial payload size (i.e., the relative increase in the size of AEs) that might be relatively high, especially for the small Android malware apps. This deficiency may cause malware detectors to be suspicious of the AEs, particularly for popular Android applications. Improving the organ harvesting used in the program slicing technique, in particular, finding the smallest vein for a specific organ, can address this limitation as each organ has usually multiple veins of different sizes.

Additionally, EvadeDroid particularly crafts malware apps to mislead the malware detectors that use *static* features for classification. We do not anticipate our proposed evasion attack to successfully deceive ML-based malware detectors that work with behavioral features specified by dynamic analysis as the perturbations are injected into malicious apps within an `IF` statement that is always `False`. Therefore, it remains an interesting avenue for future work to evaluate how our proposed attack can bypass behavior-based malware detectors.

Furthermore, since EvadeDroid uses a well-defined optimization problem outlined in Algorithm 3.1, it can be extended to other platforms (e.g., Windows) if attackers offer problem-space transformations that are tailored to manipulate real-world objects (e.g., Windows Portable Executable files). This is because the transformations used in EvadeDroid can only be applied to manipulate Android applications. We leave further exploration as future work since it is beyond the scope of this chapter.

Finally, our chapter comprehensively covers various malware detection systems, employing diverse classifiers on different features with various types. However, there is an opportunity to improve the validity of our findings since the evaluation is conducted in controlled laboratory settings. Future research should delve deeper into the applicability of our adversarial attack framework in real-world environments, where dynamic factors like evolving malware landscapes and deployment scenarios may impact the attack's performance.

3.7 Conclusions

This chapter introduces EvadeDroid, a novel Android evasion attack in the problem space, designed to generate real-world adversarial Android malware capable of evading ML-based Android malware detectors in a black-box setting. Unlike previous approaches, EvadeDroid directly operates in the problem space without initially focusing on finding feature-space perturbations. Experimental results demonstrate the effectiveness of EvadeDroid in deceiving various academic and commercial malware detectors.

3.A Problem-Space Constraints

To generate realizable AEs, adversarial attacks need to consider the following four problem-space constraints [9]:

- **Available transformations** describe the types of manipulations (e.g., adding dead codes) that an adversary can utilize to modify malware apps.
- **Preserved semantics** constraint explains that the semantics of an Android app should be maintained after applying a transformation to the app.
- **Robustness-to-preprocessing** constraint describes the requirement that non-ML methods (e.g., preprocessing operators) should not be able to undo the adversarial changes.
- **Plausibility** constraint explains adversarial apps must look realistic (i.e., naturally created) under manual inspection.

3.B Donors Evaluation

In this evaluation, we assess the influence of our donor selection strategy on the performance of EvadeDroid. Two action sets, denoted as Δ_1 and Δ_2, are provided, each containing 20 transformations. The transformations in Δ_1 and Δ_2 are chosen at random from the collection of transformations extracted from the 10 most similar apps and the 10 least similar apps to malware apps, respectively. To understand the process of finding similar apps, refer to Section 3.4.3.1. We then use these action sets in EvadeDroid to transform 50 randomly selected malware apps into AEs. Table 3.9 presents a comparison of the impact of Δ_1 and Δ_2 on EvadeDroid's performance. As can be seen in this table, when using Δ_1, the number of queries and transformations is significantly reduced compared to Δ_2. This finding validates that leveraging benign apps that resemble malware apps as the donors of transformations can reduce the cost of generating AEs, specially in terms of the required queries.

Table 3.9: The performance of EvadeDroid in attacking DREBIN when it utilizes two different action sets Δ_1 and Δ_2.

Action Set	ER	Avg. No. of Queries	Avg. No. of Transformations
Δ_1	66.0%	3	2
Δ_2	68.0%	7	3

3.C Implementation Details

The proposed framework illustrated in Figure 3.8 is implemented with Python 3 and Java 8. The source code[2] of the pipeline has been made publicly available to allow reproducibility. The components of EvadeDroid's pipeline are clearly depicted in Figure 3.8. This section

[2]https://github.com/HamidBostani2021/EvadeDroid

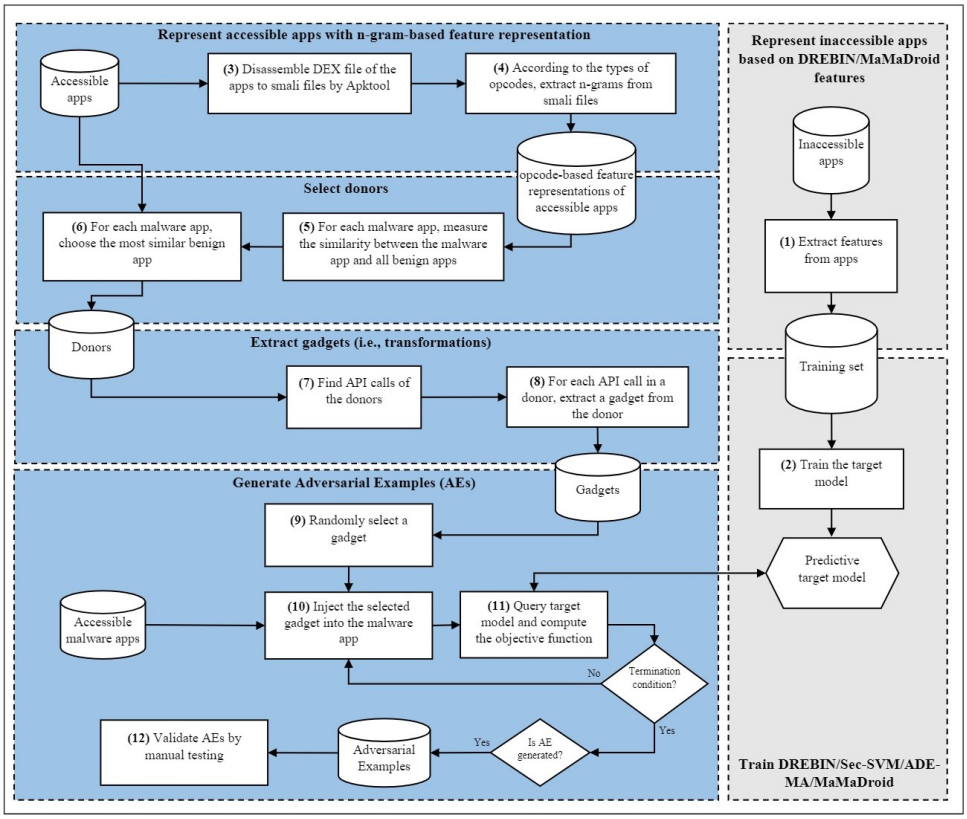


Figure 3.8: The details of the proposed framework. The blue and gray areas represent the workflows of EvadeDroid and target black-box malware detection, respectively.

reviews some of the components that have not been previously described in detail in the chapter.

- **Component 7.** To identify API calls in donor apps, we utilize the tool provided in [132]. This tool leverages Apktool [100] to access the DEX files of Android apps, which are represented as smali files. It employs string analysis techniques to scan these files and identify the API calls present within them.

- **Component 8.** We extend the tool presented in [9] to extract API calls from donors because this tool, which is based on the Soot framework, originally harvests Activities and URLs only.

- **Component 10.** The tool presented in [9] has also been used to inject gadgets into malware apps (i.e., hosts). This tool ensures the fulfillment of both the preserved-semantic and robustness-to-preprocessing constraints by utilizing opaque predicates [107] for transplanting the gadgets into hosts. The opaque predicates employed in the tool are obfuscated condition statements that encapsulate the injected gadgets. During runtime,

these statements always evaluate to `False`, thereby preserving the semantics of malware apps as the injected gadgets remain unexecuted. Furthermore, the preprocessing operators are unable to eliminate the injected gadgets as the result of the statement cannot be statically resolved from the source code during design time. It is important to note that the generated AEs are plausible. This is because the manipulation of malware apps involves the injection of realistic gadgets found in benign apps. Additionally, the injection of the gadget occurs in unnoticeable injection points, maintaining the homogeneity complexity of the host's components. The inclusion of gadgets may enhance EvadeDroid's performance by introducing more features in the manipulated apps. For further insights into the tool, we refer readers to [9].

3.D Android Malware Detectors

DREBIN [96] and **Sec-SVM** [16] are two prominent approaches in Android malware detection. DREBIN utilizes binary static features and employs linear Support Vector Machine (SVM) for classification. It extracts various features, including requested permissions and suspicious API calls, from the Manifest and DEX files of APKs through string analysis [2]. These features are then used to construct a feature space for the classifier. In DREBIN, each app is represented by a sparse feature vector, where each entry indicates the presence or absence of a specific feature. Secure SVM (Sec-SVM) is an enhanced version of DREBIN that aims to enhance the resilience of linear SVM against adversarial examples. The core concept behind Sec-SVM is to increase the cost of evading the model when generating adversarial examples. Compared to DREBIN, Sec-SVM relies on a larger set of features for malware detection, making it more challenging to evade. Since Sec-SVM is a sparse classification model, it leverages a greater number of features to improve its malware detection capabilities

ADE-MA [25] is an ensemble of deep neural networks (DNNs) that is strengthened against adversarial examples with *adversarial training*. The adversarial training method tunes the DNN models by solving a min-max optimization problem, in which the inner maximizer generates adversarial perturbations based on a mixture of attacks, i.e. iterative "max" Projected Gradient Descent (PGD) attacks.

MaMaDroid [95] utilizes static analysis to detect Android malware. The goal of MaMaDroid is to capture the semantics of an Android app by employing a Markov chain based on abstracted sequences of API calls. The process begins with generating a call graph for each Android app. From this call graph, the sequences of API calls are extracted and abstracted into different modes, including families, packages, and classes. Subsequently, MaMaDroid constructs a Markov chain for each abstracted API call in an APK, where each state represents a family, package, or class, and the transition probabilities indicate the state transitions. Finally, feature vectors incorporating continuous features are created based on the generated Markov chains.

Opcode-SVM [97] is an Android malware detection method that utilizes static opcode-sequence features instead of predefined features. This approach focuses on performing n-gram opcode analysis to represent apps in a feature space, where a malware classifier is constructed. Specifically, the method employs a linear SVM with 5-gram binary opcode features to effectively detect Android malware.

3.E Experimental Settings

Android malware detectors. We built DREBIN, Sec-SVM, MaMaDroid, and ADE-MA based on their available source codes (i.e., [133–135]) that have been published in online repositories. Moreover, we have reproduced Opcode-SVM based on the implementation details provided in [97]. The hyperparameters of the reproduced malware detectors are similar to those considered in their original studies [9, 25, 95, 97]. Note that in this chapter, the reproduced MaMaDroid [95] is based on the K-Nearest Neighbors (KNN) algorithm with $k = 5$. This malware classifier operates in the family mode in all experiments. KNN algorithm is used in MaMaDroid as we empirically concluded that KNN performs better on our dataset than other classifiers employed in [95].

Baseline evasion attacks. We implemented Sparse-RS, ShadowDroid, and GenDroid with Python 3 based on their relevant studies (i.e., [12, 13, 26]). Moreover, PiAttack [9] has been built based on their available source codes published in [133].

EvadeDroid. Besides query budget Q and the allowed adversarial payload size α that have been mentioned earlier, n is another hyperparameter that shows the length of overlapping sub-string of opcodes' types in n-gram-based feature extraction. The study conducted in this chapter considers $n = 5$ because in [97], the authors have shown that the best classification performance for opcode-based Android malware detection can be achieved with the 5-gram features. Furthermore, we select the top-100 benign apps as suitable donors for gadget extraction. Note that we consider 100 donors as organ harvesting from donors is a time-consuming process.

3.F Baseline Attacks

PiAttack [9], also known as PK-Greedy, is a white-box attack in the problem space that generates real-world AEs using transformations called gadgets. This attack comprises two main phases: the initialization phase and the attack phase. In the initialization phase, key benign features are identified, and then gadgets corresponding to the identified features are collected from benign apps. In the attack phase, a greedy search strategy is used to find optimal perturbations by selecting gadgets based on their contribution to the feature vector of the malware app. This process is repeated until the modified feature vector is classified as a benign sample. Note that PiAttack incorporates both primary features and side-effect features into malware apps. The primary features are added to bypass detection, while the side-effect features are included to meet problem-space constraints.

Sparse-RS [12] attack is a soft-label attack that gradually converts malware samples into AEs by querying the target model. Sparse-RS, which is a gray-box attack in the malware domain, finds the l_0-bounded perturbations (i.e., the maximum allowed perturbations) via random search. Note that we set initial decay factor $\alpha_{init} = 1.6$ and sparsity level $k = 180$ similar to [12] and query budget $Q = 1000$.

ShadowDroid [13] is a black-box problem-space attack that generates AEs by building a substitute classifier, which is a linear SVM. The substitute classifier is built on binary feature space compromised by permissions and API calls. This attack makes a key feature list based on the importance of features specified by the substitute classifier. The attack adds the key features to a malware app and queries the target classifier to check if the manipulated app is classified as malware. ShadowDroid continues this process until reaching the maximum

Table 3.10: The impact of various training strategies on the utility of DREBIN.

Model	No. of AEs	TPR	FPR
Standard Training	N/A	80.8%	1.7%
Adversarial Re-training	500	78.3%	1.4%
	1000	74.9%	0.9%
	1500	68.7%	0.5%
	1769	32.7%	0.2%

query budget or generating an AEs. We set query budget $Q = 100$, following a similar setting as in [13]. Note that ShadowDroid is not fully compatible with the zero-knowledge (ZK) setting as it relies on the assumption that the target detectors utilize permissions and API calls for malware detection. However, since it is a query-based problem-space attack, it serves as a proper naive problem-space baseline attack for our chapter.

GenDroid [26] is a black-box Android evasion attack building upon GenAttack [136]. This query-based attack utilizes GA to discover adversarial perturbations in soft-label settings. GenDroid extends GenAttack by redesigning the fitness function, adopting a new evolutionary strategy, and incorporating Gaussian Process Regression (GPR) to guide evolution. Specifically, the fitness function is defined through a logarithmic transformation, incorporating adjustable weight parameters (α and β) and a norm-bounded perturbation. The selection process prioritizes elite individuals with higher fitness scores, and the *Softmax* function is employed to convert fitness scores into probabilities. GPR is introduced to predict fitness values for individuals in the next generation. We empirically set the population size to 8 and the maximum number of generations to 50.

3.G Data Augmentation

In this experiment, we evaluate the performance of EvadeDroid in enhancing the adversarial robustness of Android malware detection. To achieve this, we transform malware samples from the original training set into AEs using EvadeDroid. Subsequently, we re-train DREBIN using the modified dataset, resulting in a model that is robust to EvadeDroid. Our empirical analysis demonstrates that incorporating AEs generated by EvadeDroid in the training set of DREBIN can effectively thwart the adversarial effect of EvadeDroid. However, the number of AEs employed has an effect on the DREBIN's utility (i.e., the original performance of DREBIN). Table 3.10 reveals that the addition of more AEs to the training set reduces the TPR of DREBIN. For instance, the TPR of DREBIN is reduced by 32.7% compared to the standard training when $1769K$ malware samples in the training set are transformed into AEs. It is noteworthy that out of the $2K$ malware samples in the training set, EvadeDroid is capable of generating 1769 AEs.

4

Exposing Vulnerabilities in Machine Learning for Malware Detection

Machine Learning (ML) promises to enhance the efficacy of Android Malware Detection (AMD); however, ML models are vulnerable to realistic evasion attacks—crafting realizable Adversarial Examples (AEs) that satisfy Android malware domain constraints. To eliminate ML vulnerabilities, defenders aim to identify susceptible regions in the feature space where ML models are prone to deception. The primary approach to identifying vulnerable regions involves investigating realizable AEs, but generating these feasible apps poses a challenge. For instance, previous work has relied on generating either feature-space norm-bounded AEs or problem-space realizable AEs in adversarial hardening. The former is efficient but lacks full coverage of vulnerable regions while the latter can uncover these regions by satisfying domain constraints but is known to be time-consuming. To address these limitations, we propose an approach to facilitate the identification of vulnerable regions. Specifically, we introduce a new interpretation of Android domain constraints in the feature space, followed by a novel technique that learns them. Our empirical evaluations across various evasion attacks indicate effective detection of AEs using learned domain constraints, with an average of 89.6%. Furthermore, extensive experiments on different Android malware detectors demonstrate that utilizing our learned domain constraints in Adversarial Training (AT) outperforms other AT-based defenses that rely on norm-bounded AEs or state-of-the-art non-uniform perturbations. Finally, we show that retraining a malware detector with a wide variety of feature-space realizable AEs results in a 77.9% robustness improvement against realizable AEs generated by unknown problem-space transformations, with up to 70× faster training than using problem-space realizable AEs.

This chapter is based on the published paper: H. Bostani, Z. Zhao, Z. Liu, and V. Moonsamy, Level Up with ML Vulnerability Identification: Leveraging Domain Constraints in Feature Space for Robust Android Malware Detection, ACM Transactions on Privacy and Security, 2025 [137]. The content remains unchanged from the published version.

4.1 Introduction

Due to the ongoing proliferation of Android malware, the application of Machine Learning (ML) for Android Malware Detection (AMD) continues to remain a topic of interest for security researchers [95, 96, 123, 138–143]. However, ML-based solutions are vulnerable to evasion attacks that generate adversarial examples (AEs) [144]—malware that is crafted to purposely be misclassified as benign. These adversarial attacks exploit blind spots within feature space (i.e., the model's decision space) where the decision boundary is not accurate due to insufficient training samples [29]. To confront evasion attacks, various defense strategies have been proposed to either detect (e.g., [145–147]) or eliminate (e.g., [49, 148, 149]) blind spots; nevertheless, in the Android malware domain, defenders must cope with realistic attacks that capitalize on *feasible* blind spots. Indeed, the entire blind spots do not show vulnerable regions in the context of malware because realistic evasion attacks solely target blind spots within feasible regions where feature representations of feasible apps can settle. The primary solution for uncovering vulnerable regions entails exploring realizable AEs, such as leveraging them in *adversarial hardening* [150], encompassing methods that integrate AEs into the training process. However, generating realizable AEs is challenging as AEs must be *feasible apps in the real world*, meaning they must satisfy the *domain constraints* [151, 152] (e.g., preserving malicious functionality [9]).

Generally, there are two different approaches to generating AEs. The first approach is to generate norm-bounded AEs by modifying specific features that can best mislead the detector without considering domain constraints [14–16, 18–20]. This approach is moderately efficient and can offer some degree of adversarial robustness when utilized in adversarial hardening because the feature space of realizable AEs is just a sub-space of the full space of AEs [153]. However, AEs generated by this approach in the Android domain [154, 155], might not be realizable AEs, as the resulting AE space fails to fully cover feasible regions that are vulnerable to realistic evasion attacks [150], as illustrated by Figure 4.1. For instance, the attacks proposed in [17, 24] might not always generate realizable AEs as the adversarial features added to the Manifest files of apps can be removed by pre-processing operators [9]. Figure 4.1 also indicates that exploring the full space of norm-bounded AE space is not necessary if we can model the smaller space of realizable AEs.

The second approach is to directly generate realizable AEs in the problem space, i.e., applying problem-space transformations into malicious apps that induce realizable perturbations in the feature space [9, 22, 23, 28, 84, 154]. Although this approach fundamentally ensures that the domain constraints are satisfied, we find it to be sub-optimal for adversarial hardening mainly due to three reasons. First, applying problem-space transformations is computationally expensive [156] whereas perturbing feature vectors is typically simpler and more efficient than manipulating objects in the problem space [151]. Second, finding effective problem-space transformations is challenging because they must not only mislead the detector but also meet domain constraints. For instance, the transformations utilized in [17, 28, 157] fail to meet the domain constraints because they either can be thwarted by removing newly-added content through pre-processing [17, 157] or result in functional disruptions [28]. Third, the problem-space transformations used in adversarial hardening might be ineffective to unknown attacks, implying that attackers utilizing new transformations could still successfully evade detection [156].

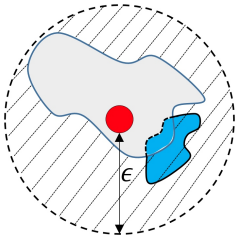


Figure 4.1: Feature space achieved by existing unrealistic attacks (blue) may not cover the realizable AE space (gray). The ϵ-ball covers all possible AEs that can be generated for the malware sample (red).

To tackle the challenges associated with identifying vulnerable regions, this chapter aims to help defenders uncover such regions by exploring the properties of feasible apps within the feature space. These properties, which represent domain constraints in the feature space, assist defenders in pinpointing feasible regions that might be susceptible to realistic evasion attacks. For instance, by leveraging feature-space domain constraints, adversarial hardening can harness the advantages of both the feature space and the problem space when generating AEs, i.e., being efficient by directly modifying features while satisfying the domain constraints. To this end, we first interpret the domain constraints of Android malware in the feature space (Section 4.3), then learn domain constraints from the feature representations of a large number of apps (Section 4.4), and finally apply them to counter evasion attacks (Section 4.5). More concretely, we first argue that Android domain constraints are meaningful feature dependencies that exist within the feature space. This implies that feature-space AEs are realizable when they adhere to these feature dependencies. Then, we introduce two sets of dependencies over the feature values, named *perfect* and *relatively strong* feature dependencies, which can represent domain constraints in the feature space. Next, we present a domain-constraint learning method to extract meaningful feature dependencies. Specifically, the proposed method utilizes statistical dependencies and Optimum-path Forest (OPF) [158] to learn domain constraints from the feature representations of training samples. Here, OPF, which is a graph-based pattern recognition method, is adapted to extract meaningful feature dependencies. Finally, we apply our learned domain constraints across various defense methods to illustrate their effectiveness. In particular, we propose an AE detection method to preemptively identify AEs by differentiating them from feasible apps using our learned domain constraints. Our empirical evaluation shows that our proposed method can successfully identify 89.6% of AEs generated by various evasion attacks. Moreover, we incorporate feature-space realizable AEs into Adversarial Training (AT) [148] to enhance the robustness of Android malware detection against realistic evasion attacks, generating problem-space realizable AEs. Such feature-space realizable AEs are generated during AT by considering not only the norm-bounded constraints but also our learned domain constraints. Our empirical analyses on DREBIN [96], DroidAPIMiner [123], and RAMDA [159], and R-PackDroid [160], four different malware detectors, reveal that our defense outperforms both AT based on norm-bounded AEs (9.3% over DREBIN, 34.5% over DroidAPIMiner, 20.0% over RAMDA, and 8.1% over R-PackDroid) and state-of-the-art AT based on non-uniform perturbations [161] (4.7% over DREBIN, 16.6% over DroidAPIMiner, 11.1% over RAMDA, and 3.1% over R-PackDroid). Our evaluation also highlights the better performance of our

defense than problem-space realizable AEs in efficiency and generalizability for adversarial retraining [162]. Our contributions[1] can be summarized as follows:

- We propose a novel interpretation of domain constraints in the feature space for AMD (Section 5.2.3). This new interpretation considers key feature dependencies of feasible Android apps in feature space to specify feasible regions where the feature representations of realizable AEs may reside. We then propose a novel domain-constraints learning technique based on the statistical correlations between features and a graph-based clustering algorithm called OPF to extract meaningful feature dependencies from large-scale data (Section 4.4.1).
- We demonstrate how these learned domain constraints can be utilized either to identify AEs, which are not feasible apps (Section 4.5.1), or to generate feature-space realizable AEs for adversarial hardening to improve the robustness of AMD against realistic evasion attacks (Section 4.5.2).
- We empirically evaluate the proposed AE detection with three evasion attacks, GenDroid [26], ShadowDroid [13], and Grosse Attack [17], demonstrating that our defense can successfully identify their generated AEs. Furthermore, our extensive experiments on four different Android malware detectors, DREBIN [96], DroidAPIMiner [123], RAMDA, and R-PackDroid [160] demonstrate that our defense provides superior model robustness than AT based on norm-bounded AEs and the state-of-the-art defense based on non-uniform perturbations [161].
- We validate both the efficiency and generalizability of our defense over adversarial retraining based on problem-space realizable AEs (Section 4.6.5).

4.2 Related Work

In this section, we first provide an overview of prior studies that have explored AEs in the Android domain, either within the feature space (Section 4.2.1) or problem space (Section 4.2.2). Furthermore, we review recent studies that explore realizable AEs with feature-space domain constraints but in domains other than Android (Section 4.2.3).

4.2.1 Feature-Space AEs

There exist a large body of related work [12, 15, 17–20, 24–26, 34] that investigated feature-space AEs. Xu et al. [26] introduced a black-box attack, incorporating the attention mechanism and the Jacobian-based saliency map algorithm. Croce et al. [12] proposed a query-based evasion attack using random search and evaluated it in various contexts, including AMD. Xu et al. [20] developed a semi-black-box framework based on the simulated annealing method to perturb features of Android apps by querying the target malware detector. Li et al. [24, 25] proposed gradient-based and gradient-free evasion attacks to generate AEs in the feature space. Rathore et al. [14] proposed two evasion attacks based on reinforcement learning to generate feature-space AEs. Liu et al. [19] used a genetic algorithm to create feature-space AEs for improving the robustness of AMD. Chen et al. [15, 18] explored different feature-space evasion attacks (e.g., anonymous attacks and well-crafted

[1] Our code is available at `https://github.com/HamidBostani2021/robust-Android-malware-detector`.

attacks) to bypass AMD. Demontis et al. [16] proposed a feature-space evasion attack to generate Android AEs by changing the features that seem important for the SVM classifier. Grosse et al. [17] generated AEs by modifying the features extracted from Manifest files of Android malware apps using a forward derivative approach. However, the above adversarial attacks might not always yield realizable AEs. In other words, AEs generated by these evasion attacks could be impossible, as they are created by only adhering to norm-bounded constraints without ensuring domain constraints i.e., available-transformations, preserved-semantic, robustness-to-preprocessing, and plausibility constraints [9]. In particular, the AEs discussed in [12, 14–20, 24–26] might not satisfy the robustness-to-preprocessing constraint because the proposed attacks considered adding features to Manifest files in order to generate AEs, while pre-processing operators can discard the added unused features [9]. Furthermore, the preserved semantic or plausibility constraints have not been thoroughly investigated in the studies mentioned above. For instance, although the authors in [24] tried to manipulate Android malware apps using feature-space perturbations while preserving the malicious functionality of the original apps, they failed to generate realizable AEs since most manipulated apps did not work.

4.2.2 Problem-Space AEs

To address the limitation of feature-space norm-bounded AEs, several studies [9, 22, 23, 28, 84, 154] have proposed different approaches for generating AEs. Specifically, they rely on problem-space transformations that satisfy domain constraints. Labaca-Castro et al. [154] generated realizable universal adversarial perturbations by applying a sequence of transformations, found by a greedy algorithm, into malware objects. Bostani and Moonsamy [84] proposed an evasion attack that gradually converts an Android malware app into an AE by leveraging transformations identified through querying the target malware detector. Cara et al. [23] crafted adversarial Android malware by injecting API calls into malware apps. Pierazzi et al. [9] proposed an evasion attack to generate real-world adversarial Android apps through problem-space transformations guided by feature-space perturbations. Chen et al. [22] used CW [163] and JSMA [164] techniques to propose an attack that can mislead AMD. Yang et al. [28] introduced two attacks named evolution and confusion attacks to present an Android evasion attack that was based on manipulating Android malware apps. Demontis et al. [16] used obfuscation to manipulate Android malware apps. It is worth noting that in the studies above, the problem-space transformations are either code transplantation (incl. harvesting slices of bytecodes extracted from benign apps) [9, 28, 84, 154], obfuscation tools [16], or dummy codes (e.g., unused API calls in Android apps) [22].

However, the practicality of utilizing problem-space transformations in adversarial hardening is debatable due to their high computational complexity [151]. It is also known to be difficult to collect diverse problem-space transformations that fully satisfy the domain constraints [84]. For instance, although Yang et al. [28] explored domain constraints in the problem space, they failed to generate realizable AEs, as their problem-space transformations caused the apps to crash, mainly because most malware apps cannot run after manipulation. Moreover, Demontis et al. [16] utilized *DexGuard*, an Android obfuscation tool, to tamper with malware apps; however, the generated AEs were unable to significantly evade the target detectors they examined, as the obfuscation techniques provided by the tool (i.e., available problem-space transformations) had a limited effect on the features critical for their detectors.

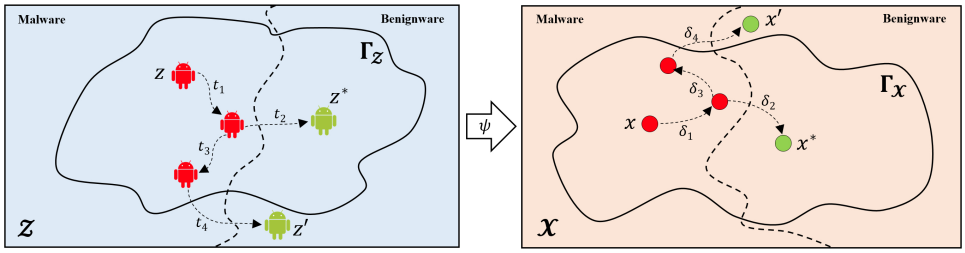


Figure 4.2: Illustration of generating AEs in the problem space Z and the feature space X where ψ shows a mapping function from Z to X. The feature-space perturbations δ_1, δ_2, δ_3, and δ_4 correspond to the problem-space transformations t_1, t_2, t_3, and t_4, respectively. The dashed lines are the decision boundaries that distinguish malware from benignware. The areas surrounded by solid closed curves represent the realizable problem space and feature space, which meet problem-space domain constraints Γ_Z and feature-space domain constraints Γ_X, respectively. z^* and x^* are realizable AEs but z' and x' are unrealizable AEs.

4

4.2.3 Feature-Space realizable AEs

Although some studies [165, 166] argue that *impossible* perturbations can still be employed to generate valid problem-space AEs if adversaries have access to the data-processing pipeline of target systems, there have been several efforts across various domains aimed at overcoming the limitations of both feature-space norm-bounded AEs and problem-space realizable AEs by generating feature-space realizable AEs. Simonetto et al. [151] introduced a generic constraint language to define feature dependencies for botnet and credit risk detection. Erdemir et al. [161] improved the adversarial robustness of DNN-based models used for malware, spam, and credit risk detection by using non-uniformed perturbations based on the PGD attack [148]. Sheatsley et al. [167] presented a formal logic framework to learn domain constraints from data used in Network Intrusion Detection Systems (NIDSs) and phishing detection. Teuffenbach et al. [168] employed domain knowledge to group flow-based features in NIDSs. Sheatsley et al. [152] proposed a domain-constraints-learning method for NIDSs based on independent features that affect other features. Chernikova et al. [169] used domain-specific dependencies (e.g., range of feature values) and mathematical feature dependencies to guarantee the realizability of AEs in NIDSs, botnet detection, and malicious domain classification. Tong et al. [170] considered so-called conserved features for improving the robustness of PDF malware detection.

Our work. To the best of our knowledge, we are the first aiming to not only thoroughly interpret how Android domain constraints (e.g., executability and plausibility of apps) are represented in the feature space for AMD but also propose a technique for learning and applying them.

4.3 Interpreting Domain Constraints in the Feature Space

In the problem space, the domain constraints of Android malware apps are defined as (i) available transformations, (ii) preserved semantics, (iii) robustness to preprocessing, and (iv) plausibility [9]; however, we aim to interpret these constraints into a set of new constraints over the feature values in the feature space. Therefore, this section introduces our novel

feature-space interpretation of domain constraints for Android malware apps. Before going into the detailed definitions (Section 5.2.3), we first provide a mathematical background of realizable AEs in both the problem and feature spaces (Section 4.3.1).

4.3.1 Problem-Space and Feature-Space Realizable AEs

Suppose $\psi : Z \to X$ is a mapping function that transforms each Android app in the problem space Z into a $d-$dimensional feature vector in the feature space X. A malware detector is an ML-based binary classifier $f : X \to Y$ with a discriminant function $h : X \times Y \to \mathbb{R}$ where $f(x) = \arg\max_{i \in Y} h_i(x)$ determines the label of $x \in X$. Specifically, $Y = \{0, 1\}$ is the label set with $y = 0$ indicating benign labels and $y = 1$ indicating malicious labels. Each element in the feature vector $x \in X$ is typically discrete [7] such as binary representations [14, 96, 123, 171–173], where 0 indicates the absence and 1, the presence of a specific feature. Generally, AEs can be generated by modifying $z \in Z$ through problem-space transformations or modifying $x \in X$ through feature-space perturbations.

Problem-Space Realizable AEs. In order to generate realizable AEs in the problem space, the following optimization is solved [9]:

$$\arg\min_{T} \quad h_1(\psi(z' = T(z))) \quad \text{s.t. } T(z) \vDash \Gamma_Z, \tag{4.1}$$

where T is a sequence of transformations that satisfy the domain constraints defined in the problem space, Γ_Z, such as preserving the malicious functionality of the malware [24].

Feature-Space Realizable AEs. In the feature space, the following optimization is solved [9, 16]:

$$\arg\min_{\delta} \quad h_1(x' = x + \delta) \quad \text{s.t. } \delta \vDash \Omega, \tag{4.2}$$

where the perturbation vector δ must satisfy the domain constraints defined in the feature space, Ω.

Most existing studies on generating feature-space AEs do not consider domain constraints but instead, adopt the naive norm bound [9] that can lead to AEs being unrealizable. In the feature space, $x' = x + \delta$ is a realizable AE if there exists *at least one* corresponding malware app z' in the problem space (i.e., $\psi(z') = x'$) that not only bypasses malware detection but also satisfies problem-space constraints Γ_Z. Figure 4.2 illustrates how adversarial transformations in the problem space make adversarial perturbations in the feature space. Reconstructing z' from x' is not possible since ψ, i.e., mapping function from Z to X, is neither *invertible* nor *differentiable* [9]. For instance, one of the main challenges in converting x', generated by a gradient-based adversarial attack, to z' arises when attempting to back-propagate the loss gradient through the mapping function that behaves like a non-differentiable layer, particularly in non-numerical domains such as malware detection [10]. While the inverse feature mapping problem [9] presents a considerable challenge in the malware domain, particularly for attackers, defenders are less impacted by this issue because their primary objective is not to generate real adversarial objects but rather to understand which adversarial perturbations are feasible within the model's decision space (i.e., the feature space). To verify the realizability of x', there is no need to directly reconstruct z' from x' to see if z' meets the domain constraints in the problem space because satisfying the domain constraints in the feature space is sufficient. In other words, x' is realizable if δ meets the domain constraints in the feature space because they demonstrate Android malware properties in the feature space.

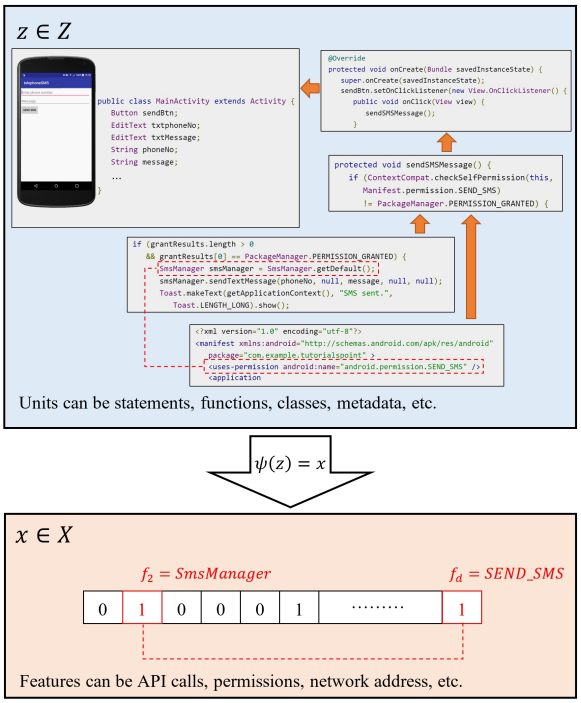


Figure 4.3: The dependency of two units in the app z is represented by the dependency of two corresponding features in the feature representation x.

4.3.2 Domain Constraints in the Feature Space

Extracting *all* feature dependencies is not only time-consuming and difficult [174] but also unnecessary because this could result in misleading dependencies (i.e., spurious correlations [48]) that have an adverse impact on robustness. Thus, we need to identify the *meaningful* feature dependencies, which can sufficiently guarantee the domain constraints in the feature space. To find out which types of feature dependencies are meaningful, we rely on predefined definitions of domain constraints that have already been formalized within the problem space [9]. Specifically, here we introduce our new feature-space interpretation for the four aspects of domain constraints defined in the problem space.

(a) Available perturbations *refer to all adversarial perturbations* $\Delta = \{\delta_1, \delta_2, ..., \delta_n\}$ *in the feature space that ensures* $x' = x + \delta_i$ *meets domain constraints. Using these perturbations makes* x' *corresponds to* ***at least one*** *problem-space realizable AE* z'.

Generally, an Android app contains different units (e.g., statements, functions, classes, and metadata) that provide various functionalities. As shown in Figure 4.3, the presence of a specific feature in the feature vector depends on the existence of the corresponding unit in the app. Moreover, the dependencies between multiple units (e.g., `SmsManager` API and `SEND_SMS` permission) also indicate that they offer a particular functionality (e.g., sending messages in Android apps). In the problem space, practical transformations are the ones

that consider these sorts of dependencies during app modification. For instance, in the code transplantation technique used to manipulate Android apps [9, 28, 84, 154], an organ (i.e., a problem-space transformation) is extracted from a donor based on the code dependencies because an organ must include all codes associated with a certain functionality [106]. In the problem space, the dependencies between units can be clarified by the *System Dependency Graph* [106]; however, these dependencies can be extracted from samples in the feature space. Therefore, we argue that using feature dependencies is sufficient for interpreting the domain constraints in the feature space. Specifically, according to the domain constraints defined in the problem space (i.e., preserved-semantic, robustness-to-preprocessing, and plausibility constraints), we introduce the following two sets of dependencies over the feature values in the feature space in (b) and (c).

(b) Perfect feature dependencies *refer to the relationships between pairs of features, signifying that both features in each pair of feature dependencies should occur together. Given a feature-space adversarial example $x' = x + \delta_i$, the perturbation $\delta_i \in \Delta$ might not satisfy domain constraints if δ_i does not guarantee all* ***perfect*** *feature dependencies.*

The semantic equivalence of two programs (e.g., Android apps) is undecidable [106], therefore, in the problem space, adversaries satisfy the preserved-semantics constraint by installing and running the manipulated app z' on an Android emulator and performing smoke testing to make sure that z' can be executed without crashing [9, 25, 28, 175]. Similarly, in the feature space, we should ensure that all *perfect* feature dependencies corresponding to an executable app also appear in x'. Otherwise, if only one of the perfectly dependent features exists in x', z' is not a feasible app, as its functionality (e.g., executability) may fail due to the lack of other dependent units.

(c) Relatively strong feature dependencies *refer to the relationships between each individual feature and the remaining features, highlighting that the feature should appear alongside at least one other feature that frequently occurs with it. Given a feature-space adversarial example $x' = x + \delta_i$, the perturbation $\delta_i \in \Delta$ might not satisfy domain constraints if each feature in δ_i does not guarantee all* ***relatively strong*** *(including perfect) feature dependencies.*

To satisfy the robustness-to-preprocessing constraint in the problem space, it is ensured that preprocessing operators cannot remove unnecessary content (e.g., unused permissions) that has been added to z during generating z' [9]. Similarly, in the feature space, we ensure that there are no removable added features appearing in x' by keeping the features that have a relatively strong dependency on each added feature. In other words, a specific feature f_j in x' is regarded as a removable feature that represents an unused unit u_j in z' when none of its dependent features appears in x'. Moreover, to satisfy the plausibility constraint in the problem space, z' is ensured to be plausible under manual inspection [9]. Similarly, in the feature space, we ensure that x' looks plausible when the feature representation is inspected.

It is noteworthy that beyond just preserving the semantics, robustness-to-preprocessing, and plausibility constraints further require the adversarial example z'/x' to be similar to a realizable app in the problem/feature space. For this reason, in the feature space, we should ensure that x' keeps all features relatively strongly dependent (i.e., the features that are not necessarily highly dependent, but whose dependencies are stronger than others) in order to achieve a similar feature representation to that of an executable Android app.

These dependencies indicate the most dependent features for each feature. Considering relatively strong feature dependencies is sufficient because they capture the most important feature dependencies. Furthermore, feature dependencies in both perfect and relatively strong feature dependencies ensure that dependent features appear together and maintain consistent relationships. This implies that any alteration in one feature must be matched by corresponding changes in its dependent feature to reflect valid and plausible combinations within the feature space. For example, if an adversary changes an app's API level from 30 to 22 while the app still requests permissions relevant only to API level 30, it creates a mismatch and results in unrealistic adversarial perturbations.

4.4 Learning Feature-Space Domain Constraints

This section introduces our learning-based method for extracting the above-defined domain constraints in the feature space. Specifically, we rely on feature correlations to identify perfect feature dependencies, and a graph-based algorithm called Optimum-Path Forest (OPF) to further identify the rest of the relatively strong feature dependencies. Regression analysis is also used in situations where it is necessary to understand how changes in dependent features affect each other. It is noted that OPF is a parameter-independent algorithm [158] that essentially considers feature dependencies in our problem to partition dependent features into a cluster.

Preliminaries of Optimum-Path Forest. OPF is an efficient pattern recognition algorithm based on graph theory [158]. This algorithm reduces a pattern recognition problem to the partitioning of a graph $G = (V, E)$ derived from input dataset [176]. G is a complete weighted graph wherein the vertices V are the feature vectors in the input dataset and the edges $E = V \times V$ are *undirected* arcs that connect vertices. Moreover, each $e_{i,j} \in E$ is weighted based on the distance between the feature vectors of corresponding vertices v_i and v_j (i.e., $d(v_i, v_j)$). OPF algorithm works based on a simple hypothesis called *transitive property* in which the vertices belonging to the same partition are connected by a chain of adjacent vertices [176]. This algorithm requires several key vertices $P \subset V$ called *prototypes* that have been found from V based on various approaches such as probability density function [177]. The OPF algorithm partitions G into different Optimum-Path Trees (OPTs) where each OPT is rooted at one of the prototypes, through a competitive process among the prototypes to conquer the rest of the vertices [176]. In general, the complete weighted graph G is partitioned into several OPTs by finding a path from each $v_i \in G$ to the best prototype $p \in P$, which provides an optimal path with the minimum path cost for v_i.

4.4.1 Our Learning Method

As depicted in Figure 4.4, our method aims to identify two types of domain constraints $\Gamma'_X = \{\Upsilon, \Lambda\}$, where Υ and Λ show perfect and relatively strong feature dependencies, respectively. The study conducted in this chapter utilizes the correlation coefficient to identify feature dependencies because, given the types of dependencies outlined in Section 5.2.3, using this measurement is sufficient for extracting our desired feature dependencies.

(a) Identification of perfect feature dependencies Υ. Based on the types of features being analyzed, a suitable correlation coefficient should be chosen to measure the correlation between every pair of features. The dependency between a pair of features is perfect if the correlation coefficient between them equals 1. We create Υ, the set of all perfect feature

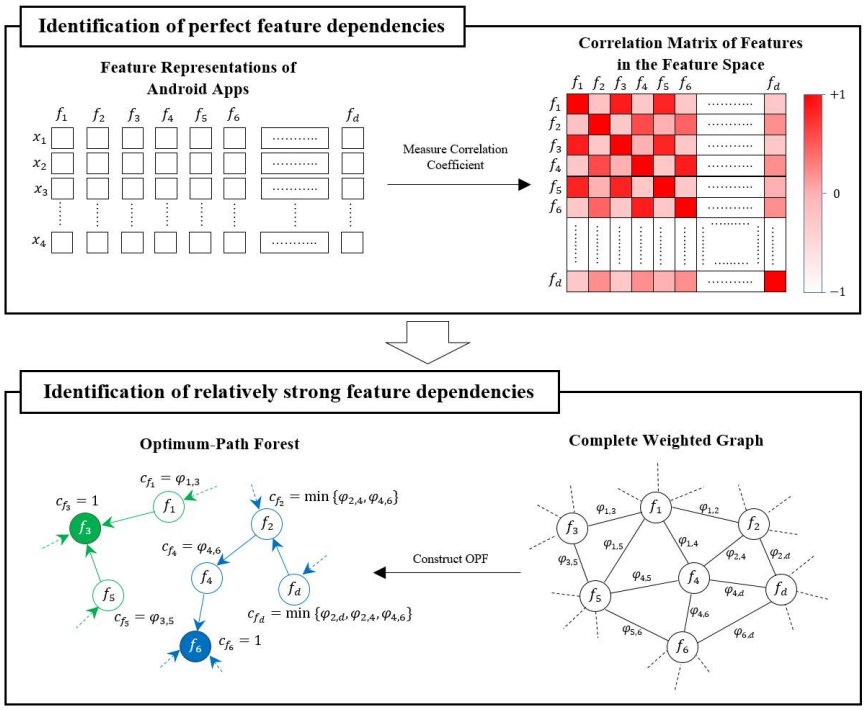


Figure 4.4: Overview of our method for learning domain constraints from data based on meaningful feature dependencies. $\varphi_{a,b}$ shows *correlation coefficient* between f_a and f_b, and c_{f_a} represents the path cost from f_a to the best prototype identified by solving equation (4.4).

dependencies based on the identified perfect correlations. Note that each $B_i \in \Upsilon$ includes all features in F that are perfectly correlated where F is the feature set of X.

(b) Identification of relatively strong feature dependencies Λ. Based on our explanation in Section 5.2.3, considering only the perfect feature dependencies is insufficient for ensuring that feature-space AEs closely resemble the feature representations of realizable Android apps. For this reason, we adopt OPF to further learn other relatively strong feature dependencies beyond the perfect ones. The proposed version of OPF partitions F into the different groups A_i where the features that are more interdependent belong to the same cluster. As shown in Figure 4.4, to construct OPF, we first create a complete weighted graph $G = (V, E)$ where $G = F$, and $E = F \times F$ includes the edges between each pair of features (f_a, f_b) weighted by correlation coefficient. Then, from each set of very strongly correlated features (i.e., $\varphi > 0.9$), we randomly select one feature as a prototype. This is due to the fact that highly correlated features can naturally indicate a potential cluster, making them suitable for clustering the remaining features. Finally, G is partitioned based on the typical method used in the OPF algorithm which is slightly modified, particularly its connectivity and cost functions, because here, the weights of edges are specified based on the correlation coefficient instead of distance as in the original algorithm. Suppose $\pi_{f_b,f_a} = \langle f_b, ..., f_k, f_a \rangle$ is a path from f_b to f_a. In the modified OPF algorithm, a connectivity function f_{min}, which is a smooth function, assigns a

path cost to each path as follows:

$$f_{min}(\langle f_b \rangle) = \begin{cases} 1 & \text{if } f_b \in P \\ -\infty & otherwise \end{cases} \tag{4.3}$$
$$f_{min}(\pi_{f_b,f_k} \cdot \langle f_k, f_a \rangle) = min\{f_{min}(\pi_{f_b,f_k}), \varphi(f_k, f_a)\}$$

where $\pi_{f_b,f_k} \cdot \langle f_k, f_a \rangle$ shows the connection of the edge $\langle f_k, f_a \rangle$ to the path π_{f_b,f_k}. As shown in equation (4.3), the path cost of π_{f_b,f_a} is the minimum weight of edges along the path. The modified OPF algorithm aims to find an optimal path for each $f_a \in F$ by maximizing f_{min} through the following cost function:

$$Cost(f_a) = \max_{\forall f_p \in P, \pi_{f_p,f_a}} \{f_{min}(\pi_{f_p,f_a})\}. \tag{4.4}$$

where P shows the prototype set. Optimum-path trees constructed by the OPF algorithm let us determine other relatively strong correlations because an OPT includes a subset of all features in the feature space (i.e., $A \subset F$) where each feature $f_a \in A$ is more dependent on other features in A as compared to the rest of features $F \backslash A$. According to the specified OPTs, we create Λ which is the set of all relatively strong feature dependencies. Each $A_i \in \Lambda$ contains all the features in F that are relatively strongly correlated.

Note that we also demonstrate that our OPF-based identification method is better than a simple baseline that uses a fixed threshold to keep highly correlated features—see results in Section 4.6.4.

Regression Analysis. For non-binary feature spaces used by some detectors like MaMaDroid [95], where its features represent Markov transition probabilities between API calls, we need to not only specify dependent features but also understand how variations in one feature affect others. Specifically, to show how adversarial perturbations in feature f_a influence feature f_b, we fit a regression model with f_a as the independent variable and f_b as the dependent variable. This model helps predict feasible values of f_b based on f_a, ensuring f_b looks representative of feasible apps.

4.5 Applying Feature-Space Domain Constraints

This section explores two approaches to demonstrate how our learned domain constraints can be applied to counter evasion attacks.

4.5.1 Adversarial Example Detection

According to our learned domain constraints, we propose a technique to identify AEs in advance, before engaging the ML model constructed for AMD. Specifically, we first introduce a way to validate how our learned feature-space domain constraints can represent the domain constraints of feasible apps. To this end, we define a new metric called *Constraints Satisfaction Rate* (CSR) to measure the ratio of the features that satisfy our learned domain constraints to all the features of a particular sample. By satisfying our learned domain constraints, we mean that one specific feature appears simultaneously with at least one of its relatively strong dependent features and all its perfectly dependent features specified in

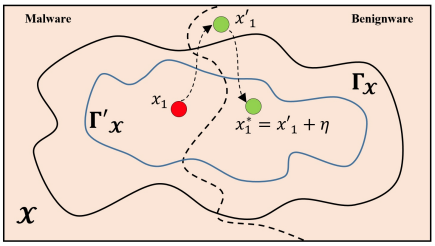


Figure 4.5: Illustration of generating a feature-space realizable AE x_1^* by adding missed meaningful dependent features η to unrealizable AE x_1'. The area surrounded by the black closed curve represents the actual realizable feature space determined by the complete domain constraints Γ_X, while the blue closed curve area represents the realizable feature space determined by our learned domain constraints Γ'_X. Our learned realizable space is a subset of the actual realizable space due to the limitation of learning from finite data.

Λ and Υ, respectively. Then, we use CSR as a criterion to distinguish AEs from feasible apps, such that an input app is considered AE if the CSR of its feature representation falls below a threshold. This criterion aids in confronting evasion attacks (e.g., [13, 17, 26]) that generate AEs through norm-bounded perturbations without considering the properties of feasible apps. For further details on these attacks, refer to Section 4.2.1.

Note that we do not expect the feature representation of feasible apps to fully satisfy our learned domain constraints because as shown in Figure 4.5, our learned domain constraints are indeed a subset of true feature-space domain constraints. This is mainly because they are learned from a finite set of samples that might not fully represent the true distribution of all existing apps.

4.5.2 Adversarial Hardening

This defense approach emphasizes integrating AEs into the training process of the classifier alongside the original training set [150]. In this chapter, we introduce two defense techniques based on adversarial hardening, which are proposed by adapting typical adversarial training [148] and adversarial retraining [162] approaches to employ our learned domain constraints.

4.5.2.1 Adversarial Training with Domain Constraints

AT is a well-established defense strategy against AEs that is widely used in the context of Android malware [14, 17, 24, 25, 28, 154]. This defense strategy proactively incorporates the generation of AEs into the training phase of ML models [178]. It solves the following min-max optimization for AT [150, 179]:

$$\min_{\theta} \mathbb{E}_{(x_i,y_i)\sim D} \quad [\max_{\delta \vDash \{\Omega, \Gamma'_X\}} \mathcal{L}(f_\theta(x+\delta), y)] \tag{4.5}$$

where L denotes the loss function and θ denotes the parameters of the Android malware detector f_θ. Moreover, $\mathbb{E}$ is the expected value of inner optimization according to $(x_i, y_i) \sim D$ indicating training data samples drawn from the distribution D. As shown in equation 4.5, the adversarial perturbations δ, which is found by solving the inner optimization, must satisfy not only initial feature space constraints (i.e., $\delta \vDash \Omega$), which is often norm-bounded

Algorithm 4.1: Applying our feature-space domain constraints

Input: **Att**.: a feature-space adversarial attack; δ: an adversarial perturbation found by **Att**.;
$\Gamma'_X = \{\Upsilon, \Lambda\}$: feature-space domain constraints.
Output: δ^*, a realizable adversarial perturbation.

1 $\delta^* \leftarrow \delta$.
2 **foreach** *feature* f_j *in* δ **do**
3 **if** *there exists* $B_k \in \Upsilon$ *including* f_j **then**
4 **foreach** *feature* $f_b \in B_K$ **do**
5 Load the regression model M for $f_j \rightarrow f_b$ if it exists; otherwise, build it.
6 Add $M(f_b)$ into δ^*.
7 **end**
8 **end**
9 Find $A_i \in \Lambda$ containing f_j
10 Select $f_a \in A_i \backslash \{f_j\}$, which appears the most important for **Att**.
11 Load the regression model M for $f_j \rightarrow f_a$ if it exists; otherwise, build it.
12 Add $M(f_a)$ into δ^*.
13 **end**
14 **return** δ^*

constraints but also our learned feature-space domain constraints (i.e., $\delta \vDash \Gamma'_X$) because as depicted in Figure 4.5, satisfying Γ'_X can turn an unrealizable AE into a realizable AE. Lines 2 to 13 in Algorithm 4.1 show how an adversarial perturbation δ becomes realizable by adding dependent features. Note that in Line 10 of Algorithm 4.1, we select a feature from $A_i \backslash \{f_j\}$ into δ^* that seems the most important for the input attack **Att**., e.g., the feature associated with the highest gradient provided by a gradient-based attack. In Lines 5 (and 11), we construct a regression model M using the training set, with f_j as the independent variable and f_b (and f_a) as the dependent variable. M is used to estimate plausible values for f_b (and f_a) based on f_j. To minimize the computational overhead of Algorithm 4.1, we cache new regression models, eliminating the need to rebuild them for similar cases. It is important to note that regression models are employed for estimating dependent feature values only when the input feature space is non-binary. For a binary feature space, it is sufficient just to add the dependent features (i.e., skip Lines 5 and 11 in Algorithm 4.1 and adjust Lines 6 and 12 to include f_b or f_a in δ^*).

Over the past years, Projected Gradient Descent (PGD) [148] has been extensively applied in the field of malware detection [24, 25, 35, 180]. The study conducted in this chapter uses PGD adapted for the Android malware domain [24] to find δ. We adopt the L_1 norm bound as the perturbation bound for PGD. This attack, which is described in Algorithm 4.2, has been modified slightly for the purpose of of this chapter. Specifically, we follow the suggestion in [24] and incorporate a normalization step to address the small-gradients issue, which may occur especially when the feature space is binary. This step involves updating the perturbation δ in the steepest gradient direction, which is computed as the unit vector e with $e_{j^*} = sign(g_{j^*})$ for $j^* = \arg\max_{1 \leq j \leq d} |g_j|$ [181], where g_j is the value of

Algorithm 4.2: PGD Attack under L_1 bounds and our feature-space domain constraints

Input: (x, y): a malware sample where x and y is the feature vector of the sample and its label; f_θ: target malware detector with parameters θ; L: loss function of f_θ; k: steps; α: step-size; q: percentile; ϵ: the L_1 norm perturbation bound; Γ'_X: our learned domain constraints; F: the feature set characterizing the dimensions of feature space X

. **Output:** x', the perturbed samples.

$\delta \leftarrow$ a vector of 0 with length x.

foreach $i = 1$ *to* k **do**

  $g \leftarrow \nabla_{\delta_i} L(f_\theta(x + \delta_i), y)$.

  $e \leftarrow \{r(g_j)$ **if** $g_j \geq P_q(g)$ **else** $0 | 1 \leq j \leq d\}$ where $r(g_j)$ is a function that rounds the value g_j up to the nearest integer, d is the dimension of x and $P_q(g)$ is the q-th percentile of g.

  $\delta_i \leftarrow \delta_{i-1} + \alpha \times e$.

  $\delta_i \leftarrow \epsilon . \frac{\delta_i}{\max\{\epsilon, \|x\|_1\}}$.

end

if *F is composed of discrete features* **then**

  $C, U \leftarrow$ Select top-ϵ features in δ with highest values, and their corresponding values.

  $\delta, \delta' \leftarrow$ two vectors of 0 with length x.

  while $\|\delta\|_1 \leq \epsilon$ **do**

    $c \leftarrow$ Remove top-1 feature from C.

    $\delta'_c \leftarrow U[c]$.

    Apply Algorithm 4.1 to ensure δ' satisfies Γ'_X.

    if $\|\delta\|_1 + \|\delta'\|_1 \leq \epsilon$ **then**

      $\delta \leftarrow \delta + \delta'$.

    end

    $\delta' \leftarrow$ a vector of 0 with length x.

  end

end

$x' \leftarrow x + \delta$.

return x'

the j-th index of the gradient $g = \nabla_{\delta_i} L(f_\theta(x + \delta_i), y)$ computed in i-th iteration of PGD, and d is the dimension of sample x. In this chapter, we consider g_j rather than $|g_j|$ since our attacker can only add features for generating AEs. Moreover, as shown in Line 4 of Algorithm 4.2, we adopt the proposed solution from [181] to address the inefficiency of updating a single feature by updating multiple features simultaneously. We use the projection operator demonstrated in [161] to project perturbation δ_i into L_1 norm bound with the size of ϵ (Line 7 in Algorithm 4.2). Note that as stated in [148], this attack lets δ be continuous during the optimization process. However, as shown in Lines 9 to 19, if the input feature space is discrete, we map δ to the discrete feature space by considering at most ϵ indices in δ

(including the top features with the highest values and their dependent features specified by Algorithm 4.1) before incorporating δ into the input malware sample. We ensure that the process of adding each feature, including its dependent features, from the top-ϵ features of the initially identified δ to the final δ is carried out in a manner that respects the perturbation bound ϵ unless the feature should not be included. Our mapping approach makes δ a realizable adversarial perturbation; however, considering only the top-ϵ indices in δ might lead to creating an unrealizable adversarial perturbation.

It is important to note that besides PGD, various other methods, especially those designed for discrete feature spaces [182, 183], can be used in the inner maximization problem of equation 4.5 to identify adversarial perturbations. In Appendix 4.A, we examine Sparse-RS [12], which was originally tested on AMD, instead of PGD, and evaluate its performance. Moreover, since we are focused on defense, the goal of Algorithm 4.1 is not to generate realizable AEs that mislead malware classifiers but rather to convert adversarial perturbations produced during robust optimization into realizable ones. Although incorporating dependent features in these perturbations may reduce misclassification confidence compared to the original adversarial perturbations, our preliminary analysis demonstrates that the misclassification confidence of the feature representations of malware samples adversarially modified by our approach is still significantly higher than that of the original malware samples, showcasing their effectiveness in AT.

4.5.2.2 Adversarial Retraining with Domain Constraints

This adversarial hardening method directly uses AEs to augment the training data. Suppose $D_m = \{(x_i, y_i)|x_i \in X, y_i = 1, i = 1, ..., k\}$ shows a fraction of all the malware samples in the input training set D. In adversarial retraining, we first construct an adversarial set $D^a_m = \{(x'_i, y_i)|x_i \in X, y_i = 1, i = 1, ..., k\}$ where each x'_i is the adversarial example of x_i that is generated by using an evasion attack. Then, we use $D' = D \cup D^a_m$, the mixed set that is augmented with AEs, to retrain the ML models.

The study conducted in this chapter considers our learned domain constraints in the evasion attack that is supposed to prepare D^a_m because we aim to augment D with realizable AEs. In fact, every adversarial perturbation δ generated by the evasion attack must satisfy both the feature-space constraints and our learned domain constraints (i.e., $\delta \vDash \Omega$ and $\delta \vDash \Gamma'_X$). Note that to adhere to our learned domain constraints, the evasion attack should add an adversarial feature alongside one of its relatively strong dependent features, as well as all its perfectly dependent features specified in Λ and Υ, respectively. The values of the newly added dependent features are determined based on the adversarial features using a regression model M, similar to the method described in Algorithm 4.2.

4.6 Experimental Results

In this section, we empirically evaluate the performance of our learned domain constraints in confronting evasion attacks aimed at tricking AMD. Specifically, our experiments aim to answer the following research questions:

RQ1. Can our learned domain constraints be applied to counter evasion attacks that generate AEs without ensuring their feasibility? (Section 4.6.2)

RQ2. Can our learned domain constraints help AT enhance the robustness of the detectors against realistic evasion attacks? (Section 4.6.3)

RQ3. Can our OPF-based method effectively extract meaningful feature dependencies? (Section 4.6.4)

RQ4. Can our feature-space realizable AEs outperform the conventional problem-space realizable AEs? (Section 4.6.5)

All the experiments have been performed on a Debian Linux workstation with an Intel (R) Core (TM) i7-4770K, CPU 3.50 GHz, 32 GB RAM, and GPU GeForce RTX 3080 Ti.

4.6.1 Experimental Setup

Threat Models and Attacks. The evasion attacks considered in our experiments generate AEs based on the threat model that is described with three attributes:

- **Adversary's Goal.** The goal of the adversary is to trigger the Android malware detector to misclassify the adversarial (malware) example as benign.
- **Adversary's Knowledge.** The adversary may have perfect knowledge (PK), limited knowledge (LK), or zero knowledge (ZK) about the target model, including its learning algorithm, training data, feature space, and parameters. In other words, in PK, LK, and ZK attacks, the target model is considered as a white-, gray-, and black-box model by the adversary, respectively. In our experiments, we assess the efficacy of our approach using whole three types of attacks.
- **Adversary's Capability.** The adversary can generate AEs either in the feature space by perturbing feature representations under feature-space constraints, or in the problem space by applying a sequence of transformations under domain constraints [9]. Here we follow the common practice of only considering feature-addition transformations/perturbations [9, 17, 84].

To explore **RQ1**, we use three evasion attacks, GenDroid [26], ShadowDroid [13], and Grosse attack [17], which generate AEs under ZK, LK, and PK settings, respectively. These attacks generate adversarial perturbations irrespective of domain constraints, raising doubts about the realizability of the resulting AEs. Moreover, our work considers a realistic problem-space attack, known as *PiAttack*[2] [9], to empirically investigate different RQs stated in Section 4.6. PiAttack is a white-box attack that generates problem-space realizable AEs by applying effective problem-space transformations (i.e., code snippets called gadgets extracted from donor apps) specified by feature-space perturbations. Note that PiAttack can be regarded as an adaptive attack, as it inherently knows our domain constraints. We refer the reader to Appendix 4.B for the technical details of PiAttack.

As a comparison, we also consider the well-known PGD attack introduced in Section 4.5, which directly adds perturbations in the feature space constrained by specific L_1 norm bounds. Specifically, we use PGD for both AT and attacking target detectors. Furthermore, in the experiments involving adversarial retraining (i.e., Section 4.6.4 and Section 4.6.5), we have

[2]PiAttack is also referred to as the PK-Greedy attack.

developed a feature-space attack called PK-Feature, which operates in a manner similar to PiAttack within the feature space. See Appendix 4.C for more details about this attack.

Target Detector. We use the well-known DREBIN-Support Vector Machine (SVM) [96] as a baseline target Android malware detector. This detector builds a linear SVM in a binary feature space consisting of eight types of features (e.g., permissions and restricted API calls). The regularization hyperparameter of the implemented linear SVM is $C = 1$ [9]. It is worth mentioning that for AT, we also consider DREBIN-Deep Neural Network (DNN) [154] consisting of four layers with the dimensions as $10,000 \times 1,024 \times 512 \times 2$. We also take into account another DNN-based malware detector called DroidAPIMiner-DNN. This malware detector, which leverages the DroidAPIMiner [123] feature representation (i.e., a binary feature representation solely constructed from API calls appearing in Android apps), consists of four layers with the dimensions as $337 \times 256 \times 128 \times 2$, where 337 represents the dimension of our samples represented based on DroidAPIMiner feature representation. Given the potential limitations in DREBIN and DroidAPIMiner, as discussed in [184] and [185] respectively, we consider a more advanced malware detector called RAMDA-DNN. This new malware detector, built upon a robust feature representation derived through Autoencoder [159], exhibits an architecture similar to DroidAPIMiner but instead uses 269 in the input layer, showing the dimensions of samples. All DREBIN-DNN, DroidAPIMiner-DNN, and RAMDA-DNN employ ReLU activation functions in their hidden layers and a Sigmoid activation function in their output layer. We train these malware classifiers for 100 epochs with a batch size of 1024.

While binary feature representation is common in the malware domain, we consider another family of malware detectors called R-PackDroid-DNN, which operates on a discrete, non-binary feature representation to demonstrate the broad applicability of our method. This detector utilizes the R-PackDroid-DNN [160] feature representation, where each feature represents the frequency of a specific type of system API component, such as a package, class, or method, within Android apps. The R-PackDroid-DNN model is composed of four layers with dimensions $2155 \times 1024 \times 512 \times 2$, where 2155 corresponds to the dimensionality of the samples, based on the R-PackDroid feature representation derived from the classes of system API components employed in the apps.

Dataset. We use a public Android dataset [9] including $\approx 152K$ Android apps collected from AndroZoo [125]. In this dataset, an app is defined as malware if it is detected by 4+ VirusTotal AVs, and as a benign sample, if no AVs detect it. We randomly select a test set of $30K$ samples, comprising $25K$ benign and $5K$ malware samples, while the remaining $\approx 122K$ samples, including $\approx 111K$ benign and $\approx 11K$ malware samples, are designated as the training set. Note that we consider a fair proportional distribution between benign and malware samples to mitigate spatial bias [128]. To assess DREBIN, samples are represented based on the DREBIN feature representation [96], a widely used binary feature set in recent studies [9, 12, 20, 24, 84, 154]. Since the DREBIN feature space is a very high-dimensional (i.e., over $1M$ features) but sparse feature space, with a significant amount of redundant features impacting DREBIN's performance [184], we select the $10K$ most distinguishing features following the recommendations from previous studies [16, 25]. We perform feature selection and feature dependency extraction solely on the training set, preventing the data-snooping pitfall [48]. To measure the correlation between every pair of features, which is crucial for extracting meaningful feature dependencies, we use *phi*

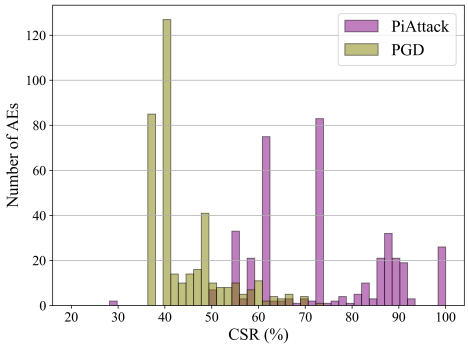


Figure 4.6: CSR of AEs generated by the domain-constraint-aware attacks, PiAttack vs. the domain-constraint-agnostic attacks, PGD, when they target DREBIN-DNN.

coefficient [186] because the feature space of target detectors considered in our evaluation consists of binary features. To evaluate the adversarial robustness of different Android malware detectors, we randomly select $1K$ malware samples from the test set, representing diverse malware families, to generate the AEs.

Evaluation Metrics. For evaluating the malware detectors, we consider Accuracy (Acc) as well as True Positive Rate (TPR) and False Positive Rate (FPR). Specifically, we calculate *clean Acc* on benign and malware examples for model utility and *robust Acc* on adversarial malware examples for robustness.

4.6.2 Evaluating Our Learned Domain Constraints

To answer **RQ1**, we first validate the utility of our learned domain constraints for representing Android malware properties by empirically evaluating if they can help to distinguish realizable AEs from unrealizable AEs. To this end, we demonstrate how the added features in AEs generated by different attacks on the DREBIN-DNN, which is based on standard training, can satisfy our learned domain constraints. Specifically, we consider two attacks, i.e., the domain-constraint-aware attack, PiAttack, and the domain-constraint-agnostic attack, PGD. Both attacks are bounded by the L_1 norm $\epsilon = 30$. We calculate CSR defined in Section 4.5.1 for the AEs successfully generated by both attacks. Figure 4.6 demonstrates that AEs generated by the realistic attack can better satisfy our learned domain constraints than the unrealistic attack. Specifically, the average CSR of AEs generated by PiAttack is 73.8%, and that of the AEs generated by PGD is only 44.6%. The relatively high CSR results related to PiAttack confirm that our extracted feature dependencies can adequately represent the domain constraints. Note that we expect that using a larger number of training samples would further improve the results.

Our empirical evaluation suggests that it is possible to differentiate realizable AEs from unrealizable AEs by setting a CSR threshold. Specifically, we consider the AEs above this CSR threshold as realizable AEs and otherwise, unrealizable AEs. Here we calculate the CSR for each example based on all its features rather than only the added adversarial features because, in practice, it is not known which of all features are added by an attack. Figure 4.7 shows the results with varied CSR thresholds and ϵ bounds. As can be seen, the

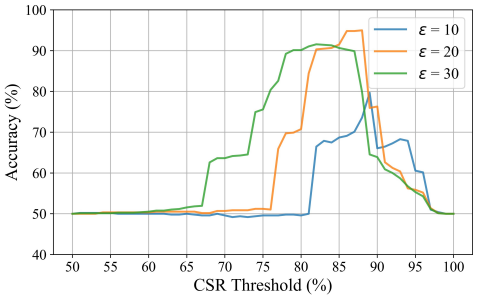


Figure 4.7: Differentiating realizable AEs generated by PiAttack from unrealizable AEs generated by PGD based on our learned domain constraints when the attacks target DREBIN-DNN. Results are reported for various L_1 norm bounds.

CSR threshold should be high enough to differentiate realizable AEs from unrealizable AEs. However, the CSR should not be set too high because our learned domain constraints may not perfectly represent the actual domain constraints. In addition, the differentiation is easier when the ϵ bounds are high (e.g., $\epsilon = 20$ and $\epsilon = 30$). Note that a lower ϵ requires a higher threshold for optimal accuracy. This is because a lower ϵ implies fewer perturbed features, which means more features remain the same as the original features, leading to a higher CSR baseline.

Finally, we assess our defense, introduced in Section 4.5.1, to show the impact of our learned domain constraints on identifying AEs generated by three evasion attacks: GenDroid, ShadowDroid, and Grosse Attack, whose resulting AEs may not be realizable. Specifically, we prepare an evaluation set comprising $5K$ benign samples and $1K$ malware samples randomly selected from the test set. Depending on the attack under investigation, AEs of malware samples generated by GenDroid, ShadowDroid, or Grosse Attack targeting DREBIN-DNN are also included in the evaluation set. All attacks are constrained by an L_1 norm bound of $\epsilon = 30$ for AE generation. Furthermore, our preliminary evaluation indicates that setting the CSR threshold at 92% can result in high detection rates of AEs. Table 4.1 demonstrates the capability of our proposed AE detection method to effectively identify AEs. It is worth noting that utilizing a smaller CSR threshold can decrease the occurrences of false detections, albeit at the expense of reducing AE detection rates.

Table 4.1: TPR and FPR of the proposed AE detection method against various evasion attacks.

Attack	**TPR**	**FPR**
GenDroid	97.5%	6.9%
ShadowDroid	95.2%	4.3%
Grosse Attack	75.4%	6.9%

RQ1. Can our learned domain constraints be applied to counter evasion attacks that generate AEs without ensuring their feasibility?
Yes, because our learned domain constraints are capable of distinguishing between realizable AEs and unrealizable AEs.

4.6.3 Evaluating Our Defense

In order to answer **RQ2** stated in Section 4.6, this section empirically evaluates our defense, which is based on AT with realizable AEs generated by considering feature-space domain constraints, as introduced in Section 4.5.2.1. Specifically, DREBIN-DNN, DroidAPIMiner-DNN, RAMDA-DNN, and R-PackDroid-DNN are trained with different strategies: standard training indicating w/o defense, AT with unrealizable AEs, and our AT with realizable AEs. We also consider a state-of-the-art AT strategy [161] that relies on non-uniform perturbations, denoted as AT-Non-Uniform-Perturbations in our experiment. We refer the reader to Appendix 4.D for further details about the AT-Non-Uniform-Perturbations approach. For all three AT approaches, PGD [24] is adopted to generate AEs in every training epoch. Specifically, half of the training malware samples are used to generate AEs and the rest remain unmodified. To generate realizable AEs in the R-PackDroid feature space, we use random forest regression, which can capture the complex relationships, to build the regression models mentioned in Algorithm 4.1. Our preliminary evaluation shows that PiAttack requires adding an average of 30 new features to achieve a successful AE in attacking DREBIN-DNN, 6 new features for DroidAPIMiner-DNN, and 15 new features for both RAMDA-DNN and R-PackDroid-DNN. Therefore, we adopt L_1 norm bound with $\epsilon = 30$ for DREBIN-DNN, $\epsilon = 6$ for DroidAPIMiner-DNN, and $\epsilon = 15$ for both RAMDA-DNN and R-PackDroid-DNN. It is worth emphasizing that the perturbation bounds for all AT approaches are the same. Moreover, since AT uses minibatch Stochastic Gradient Descent to train DNN models, there is inherent randomness in the selection of samples, particularly malware samples, in each batch. To account for this randomness and reduce bias, we perform the experiment across five trials. This involves repeating the experiments five times and reporting the average results for both the model's performance on clean data and its adversarial robustness.

Table 4.2 shows the performance of different detectors on clean samples and Figure 4.8 reports their robustness. For robustness, we test both the unrealistic attack, PGD, and the realistic attack, PiAttack, and vary the ϵ values for both attacks. Considering large perturbation bounds beyond norm-bounded perturbations can provide insights into the detector's performance against realistic attacks, which normally may succeed with large perturbations. We make sure that the AEs are generated from the malware samples that were correctly detected by malware detectors. As can be seen, in general, different defenses yield similar clean performance (Table 4.2) but different robustness (Figure 4.8). For DREBIN-DNN, our defense achieves the best robustness with an accuracy of 92.7%, thus surpassing the performance of AT-Unrealizable AEs and AT-Non-Uniform-Perturbations for $\epsilon = 30$. Specifically, for larger values of ϵ, the improvement of our proposed approach over other robust detectors, especially over AT-Non-Uniform-Perturbations slightly increases. For DroidAPIMiner-DNN, RAMDA-DNN, and R-PackDroid-DNN, our defense is still the best, e.g., 59.9% vs. 43.3% for AT-Non-Uniform-Perturbations at $\epsilon = 6$ when the target detector is DroidAPIMiner, 73.5% vs. 62.4% for AT-Non-Uniform-Perturbations at $\epsilon = 15$ when the

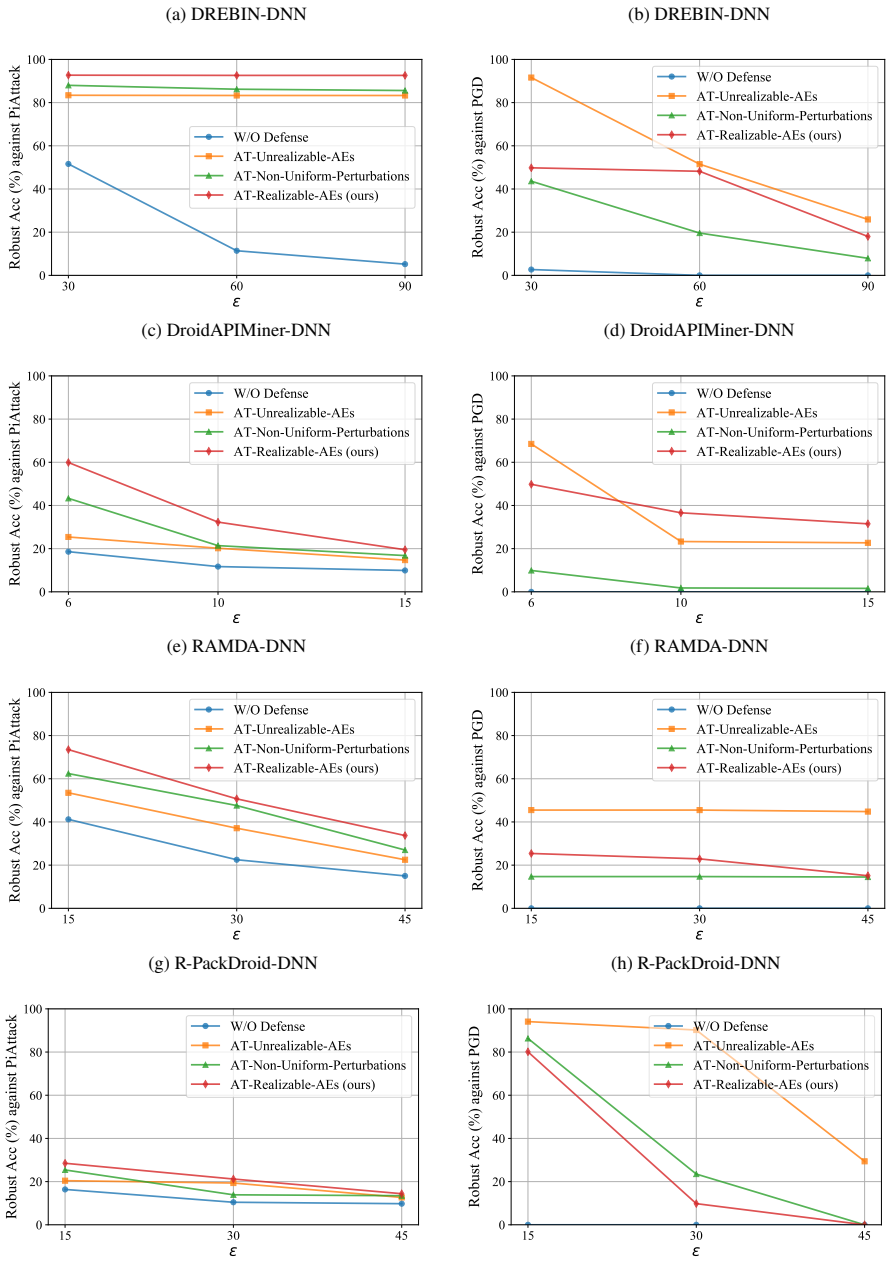


Figure 4.8: The adversarial robustness of different detectors against both an unrealistic attack (PGD) and a realistic attack (PiAttack). Results are averaged over five trials. DREBIN-DNN, DroidAPIMiner-DNN, and RAMDA-DNN operate on binary feature spaces, whereas R-PackDroid-DNN operates on a non-binary feature space.

Table 4.2: The average model utility of different malware detectors based on five trials. DREBIN-DNN, DroidAPIMiner-DNN, and RAMDA-DNN operate on binary feature spaces, whereas R-PackDroid-DNN operates on a non-binary feature space.

Detector	Defense	TPR	FPR	Clean Acc
DREBIN-DNN	W/O Defense	81.0%	0.4%	96.4%
	AT-Unrealizable-AEs	79.6%	0.4%	96.2%
	AT-Non-Uniform-Perturbations	81.2%	0.4%	96.5%
	AT-Realizable-AEs (ours)	81.0%	0.4%	96.3%
DroidAPIMiner-DNN	W/O Defense	77.2%	1.0%	95.4%
	AT-Unrealizable-AEs	76.1%	1.0%	95.2%
	AT-Non-Uniform-Perturbations	74.5%	1.0%	94.9%
	AT-Realizable-AEs (ours)	75.1%	1.0%	95.0%
RAMDA-DNN	W/O Defense	86.1%	0.9%	96.9%
	AT-Unrealizable-AEs	84.9%	0.8%	96.8%
	AT-Non-Uniform-Perturbations	85.3%	0.9%	96.7%
	AT-Realizable-AEs (ours)	84.0%	0.9%	96.6%
R-PackDroid-DNN	W/O Defense	80.4%	0.9%	95.9%
	AT-Unrealizable-AEs	77.7%	1.1%	95.4%
	AT-Non-Uniform-Perturbations	77.4%	1.0%	95.4%
	AT-Realizable-AEs (ours)	78.0%	1.0%	95.5%

target detector is RAMDA-DNN, and 28.5% vs. 25.4% for AT-Non-Uniform-Perturbations at $\epsilon = 15$ when the target detector is R-PackDroid-DNN. Figure 4.8 also demonstrates that evaluating detectors against unrealistic attacks may not accurately reflect the actual robustness against realistic attacks. As an example, with DREBIN-DNN as the target detector at $\epsilon = 30$, it would slightly overestimate the robustness of AT-Unrealizable AEs but largely underestimate the robustness of AT-Non-Uniform-Perturbations and our AT-Realizable AEs.

RQ2. Can our learned domain constraints help AT enhance the robustness of the detectors against realistic evasion attacks?
Yes, incorporating our learned domain constraints provides AT with feature-space realizable AEs, which are more effective than AT with feature-space unrealizable AEs.

4.6.4 Evaluating Our OPF-based Method

In this experiment, we aim to answer **RQ3** stated in Section 4.6 by validating the ability of OPF to extract meaningful feature dependencies. We compare the proposed OPF-based method with a straightforward baseline method that is based on threshold clustering (TC). For a specific feature f_a, TC exclusively chooses the dependent features from its top-N dependent features. As a sanity check, we also report the results for TC with bottom-N dependent features, where the least dependent features are used. This experiment specifically compares the adversarial robustness of DREBIN-SVM retrained based on the defense introduced in

Section 4.5.2.2 by changing 500 malware samples of the training data set to AEs when OPF- or TC-based approach is used to make them realizable. Note that for re-training DREBIN-SVM with AEs, we use PK-Feature under our learned domain constraints. Here we measure the robustness of different augmented DREBIN-SVMs against transferable problem-space realizable AEs generated by PiAttack using the original DREBIN-SVM as the surrogate detector. This is because generating problem-space AEs directly on different augmented DREBIN-SVMs in a white-box setting is time-consuming. Moreover, we make sure that the AEs are generated from the malware samples correctly detected by all augmented DREBIN-SVMs, and the results are calculated on successful AEs for the original DREBIN-SVM.

Table 4.3 shows that the clean accuracy for all three detectors is comparable. For robustness, although the TC method confirms the effectiveness of considering feature correlations in identifying feature dependencies that enhance adversarial robustness, our proposed OPF method surpasses it (82.9% vs. 69.4% under Top-80). In contrast, as expected, using bottom-N features leads to significantly worse results. It is also worth mentioning that the TC method requires careful tuning (e.g., via linear search) of the hyperparameter N to achieve the best performance but our OPF approach does not.

RQ3. Can our OPF-based method effectively extract meaningful feature dependencies?
Yes, OPF notably outperforms the baseline that simply selects top-N dependent features.

4.6.5 Feature-Space Realizable AEs vs. Problem-Space Realizable AEs

To explore **RQ4** stated in Section 4.6, we assess how using different types of realizable AEs impacts the training speed and generalizability of ML-based malware detection. Note that

Table 4.3: The model utility and robustness of DREBIN-SVM augmented by OPF-based method vs. TC-based method.

Detector	N	Clean Acc	Robust Acc
OPF-DREBIN-SVM	N/A	96.6%	82.9%
TC-DREBIN-SVM (Top-N)	20	96.6%	45.7%
	40	96.7%	64.6%
	60	96.6%	65.2%
	80	96.8%	69.4%
	100	96.7%	57.1%
TC-DREBIN-SVM (Bottom-N)	20	96.7%	12.9%
	40	96.7%	13.8%
	60	96.7%	24.0%
	80	96.6%	13.6%
	100	96.6%	11.6%

when investigating generalizability, we aim to understand how well a detector, enhanced with realizable AEs, can resist problem-space attacks that generate realizable AEs distinct from those used during model training. In other words, our objective is to evaluate the robustness of a detector hardened with certain transformations against attacks utilizing new transformations. To accomplish this, we retain some problem-space transformations exclusively for the purpose of attacking, while others are available for use during the training phase. Specifically, we randomly select a subset of collected problem-space transformations. We then employ PiAttack with this subset of transformations to convert 500 malware samples in the training set into problem-space realizable AEs. Subsequently, we expand our training set by incorporating these problem-space realizable AEs and proceed to retrain DREBIN-SVM. We follow a similar process to retrain DREBIN-SVM with our feature-space realizable AEs, which are generated by PK-Feature operating under our learned domain constraints, in contrast to the unconstrained variant (PK-Feature-un) that uses perturbations without enforcing such constraints. From Table 4.4, we can observe how various retraining strategies perform in terms of training speed and the level of robustness they offer. PiAttack directly targets DREBIN-SVM hardened with different defenses to generate problem-space realizable AEs from a set of 500 malware apps. In Table 4.4, NoF indicates the number of AEs, R_Time denotes the retraining time, and C_Acc represents the clean accuracy. Moreover, R_Acc1 and R_Acc2 denote robust accuracy against realizable AEs generated by a subset of problem-space transformations, which are also employed during retraining, and against all problem-space transformations, respectively. Table 4.4 demonstrates the significant increase in computational complexity when retraining DREBIN-SVM with problem-space realizable AEs compared to the scenario where we utilize feature-space AEs for retraining.

Furthermore, it's evident that AT-PiAttack primarily enhances robustness against realizable AEs that resemble those used during retraining. However, this robustness significantly diminishes when faced with realizable AEs that might differ from those employed in adversarial retraining. On the other hand, employing feature-space realizable AEs during retraining aids in maintaining the detector's robustness. In fact, utilizing feature-space domain constraints enables us to rapidly generate a greater number of realizable AE variants from each malware sample, thereby uncovering more vulnerable regions during the retraining process. As depicted in Table 4.4, our AT-PK-Feature defense, which retrains DREBIN-SVM with $\approx 10K$ feature-space realizable AEs, exhibits strong robust accuracy against PiAttack when it can utilize all available transformations. It is worth noting that generating a large number of additional (problem-space) realizable AEs is not practical for AT-PiAttack due to its high computational cost.

RQ4. Can our feature-space realizable AEs outperform the conventional problem-space realizable AEs?
Yes, feature-space realizable AEs yield higher AT training efficiency and generalizability of the detector.

4.6.6 Discussion

In order to improve the adversarial robustness of malware detection, it is necessary to provide a realistic view of the vulnerabilities of ML-based detectors to realizable AEs, which are

Table 4.4: The performance of different defenses used in DREBIN-SVM in terms of re-training time, clean and robust accuracy.

Defense	NoA	R_Time	C_Acc	R_Acc1	R_Acc2
AT-PiAttack	500	110,543s	96.6%	95.6%	5.0%
AT-PK-Feature (ours)	500	224s	96.7%	94.2%	20.8%
AT-PK-Feature-un		187s	96.7%	83.2%	14.8%
AT-PK-Feature (ours)	$\approx 10K$	1,559s	96.7%	100.0%	82.9%
AT-PK-Feature-un		952s	96.6%	85.4%	37.4%

generated under domain constraints of malware apps [154]. Our experimental results derived from various experiments have demonstrated the general effectiveness of our feature-space solution in hardening Android malware detection against evasion attacks. Note that our experiments are designed to empirically assess the efficacy of different aspects of our feature-space solution. The proposed solution is structured around a three-fold approach, which includes (1) establishing feature-space domain constraints by analyzing various aspects of Android malware properties, (2) learning feature-space domain constraints from large-scale data, and (3) applying the learned feature-space domain constraints to fortifying the robustness of AMD against evasion attacks. Specifically, our analysis in Section 4.6.2 and Section 4.6.4 validates the performance of our solution for the first and second folds. Moreover, our assessment in Section 4.6.2, Section 4.6.3, and Section 4.6.5 provides compelling evidence in support of the third fold. Here we further discuss the advantages of our solution from three main aspects: practicality, generalizability, and detectability.

Practicality. Our findings in Section 4.6.3 and Section 4.6.5, indicate that generating feature-space AEs to incorporate in adversarial hardening can be a promising alternative to generating time-consuming problem-space AEs, particularly when they successfully satisfy domain constraints. Generally, PiAttack requires an average of 5 problem-space transformations to convert a malware app into a problem-space AE. Each transformation involves several steps: (i) choosing a gadget (i.e., a slice of an app's bytecode) that indicates a problem-space transformation and loading it from disk, (ii) injecting the gadget into the malware app using Soot [101], (iii) performing static analysis using Apktool [100] (i.e., a reverse engineering tool), and (iv) constructing the feature representation based on the extracted features in the feature space of the malware classifier. Although all of these steps involve time and computational overhead, gadget injection and static analysis are particularly time-consuming compared to other processes, each averaging around 20 seconds. In the PiAttack, memory overhead is also an important consideration alongside computational time. Each step involved in transforming a malware app into an adversarial app contributes to the overall memory usage as follows:

- Load and inject the gadget: Loading a gadget from the disk and injecting it into the app requires additional memory. The size of the gadget and its integration with the app's existing codebase can impact memory consumption. For instance, loading a 1 MB gadget into memory might require slightly more than 1 MB due to overheads associated with loading and managing data structures.

- Static analysis: performing static analysis with tools like Apktool involves parsing and analyzing the app's APK file. This process can be memory-intensive, as the tool needs to maintain a representation of the app's structure and resources in memory during analysis. For instance, for a 5 MB app, the static analysis tool could require around 15 MB (5 MB app size × 3 for analysis overhead).

Generating AEs in the feature space eliminates the need to conduct steps (i) to (iv) for each modification, thereby significantly accelerating the speed of AT. As shown in Table 4.4, the retraining time for DREBIN-SVM is 110,543 seconds when using 500 problem-space realizable AEs for hardening. In contrast, the retraining time is only 224 seconds with 500 feature-space realizable AEs. Specifically, our empirical analysis in Table 4.5 demonstrates that creating a feature-space realizable AE in a binary features space takes only 0.44 seconds, whereas generating a problem-space realizable AE takes 221.09 seconds. This remarkable improvement is attributed to the fact that our approach eliminates the time-consuming process of generating problem-space AEs based on transformations. Our analysis reveals that generating feature-space AEs in a non-binary feature space takes around 0.81 seconds, which is still significantly quicker than the 221.09 seconds needed to generate problem-space AEs. Specifically, constructing the regression model, the most time-consuming phase in generating feature-space AEs, requires about 0.03 second with random forest regression using 10 decision trees on a 120K sample training set. This time is negligible when compared to the significant computational effort needed for problem-space AEs. Additionally, building a regression model is not always required, as a pre-constructed model for certain independent and dependent features may already be available, as outlined in Algorithm 4.1. In summary, the significant improvement in generating AEs within the feature space during AT indicates the potential use of feature-space realizable AE in real-world scenarios where defenders need to harden large Android malware detectors with AT.

It noted that in the preprocessing stage, we need to measure the correlation between each pair of features in the dataset. For low-dimensional datasets like DroidAPIMiner, this process takes only a few seconds. However, for very high-dimensional datasets like DREBIN, it can take over 10 hours on our Debian Linux workstation equipped with an Intel (R) Core (TM) i7-4770K CPU at 3.50 GHz and 32 GB of RAM. Despite the longer preprocessing time for high-dimensional datasets, this process is still significantly more efficient than the processing required for problem-space attacks. For example, collecting around 500 problem-space transformations, including program slicing of gadgets from benign apps, can take more than a week.

Table 4.5: Computational time required to generate AEs using various attacks.

Attack	Attack Surface	Avg. Time for Generating AE	Avg. Time per Modification
PiAttack	Problem space	221.09s	44.2s
Our approach	Binary feature space	0.44s	0.09s
	Non-binary feature space	0.81s	0.16s

Table 4.6: The clean and robust accuracy of different defenses used in DREBIN-DNN on old/new test samples. The new samples have been released between 2020 and 2022. The hardened models are strengthened using PGD with $\epsilon = 30$.

Defense	Clean Acc		Robust Acc	
	Old Test Set	New Test Set	Old Test Set	New Test Set
W/O Defense	96.4%	89.1%	51.6%	32.0%
AT-Unrealizable-AEs	96.2%	88.1%	83.4%	73.2%
AT-Non-Uniform-Perturbations	96.5%	86.7%	88.0%	77.4%
AT-Realizable-AEs (ours)	96.3%	85.2%	92.7%	80.2%

Generalizability. As examined in Section 4.6.5, our feature-space realizable AEs exhibit high generalizability as there are no limitations on the generation of realizable AEs in the feature space. On the other hand, generating problem-space realizable AEs has specific limitations because they rely on limited sets of problem-space transformations [17, 28, 157]. This means it is possible that a realistic attack based on a new set of transformations can bypass the detector that is adversarially trained on those limited sets [154]. In contrast, feature-space realizable AEs can potentially generate more diverse realizable AEs when they take into account the domain constraints in the feature space. For instance, our empirical analysis shows that for each problem-space realizable AE generated by PiAttack, we can generate about 20 different variants of feature-space realizable AEs using PK-Feature attack under our learned domain constraints.

Detectability. The results outlined in Section 4.6.2 illustrate that our learned domain constraints are able to function as a preprocessing method in detecting AEs prior to engaging an ML-based malware detector. Specifically, this non-ML defense entails a thorough analysis of the feature representations of apps to uncover suspicious apps that are not practically feasible by determining their violations from the predefined domain constraints.

4.7 Limitations and Future Work

While we have demonstrated the effectiveness of our proposed defensive approach through extensive experiments, there are some limitations that need further investigation.

First, the proposed technique can be applied to various ML-based Android malware detectors, but it may prove more effective for those utilizing expressive features, such as domain-specific features (e.g., API calls and permissions) that capture complex relationships within Android apps. Detectors that rely on more basic features, such as byte sequences [187] or opcode analysis [97], may not fully benefit due to the lack of rich dependencies. Therefore, our approach could have a greater impact on a subset of Android malware detectors that leverage more expressive feature representations. Future work could explore adapting our method to more basic feature representations.

Second, like any data-driven technique, our approach may be potentially biased toward the specific training data because data might be inaccurate and incomplete. For instance, as shown in Table 4.6, the purely data-driven approaches might intensify the *concept drift* issue [188], which is a common challenge in the ML context. Here, we empirically illustrate this limitation by testing different DREBIN-DNNs introduced in Section 4.6.3 on $15K$ ($12K$

benign and 3K malware) newly collected Android apps from AndroZoo [125]. These new samples have been released between 2020 and 2022. As can be seen from Table 4.6, all of the newly measured clean accuracies are reduced compared to the old results, indicating the existence of concept drift. While concept drift can also affect adversarial robustness, Table 4.6 demonstrates that the proposed approach still achieves better robust accuracy than other detectors, confirming its effectiveness in providing adversarial robustness against newer adversarial malware apps.

Our study in this chapter shows that feature correlations are theoretically adequate for identifying meaningful dependencies that reflect domain constraints in the feature space, but their success depends on the quality of the training data. Feature correlations can perform well when the utilized dataset accurately represents the true data distribution. However, as indicated in Table 4.6, correlations may vary if the dataset does not reflect the true data distribution. To address this concept drift, techniques like continuous learning can help maintain accuracy by adapting to current distributions of malware and benign apps. Incorporating domain knowledge could also address this limitation. However, a key question is how the knowledge of domain experts can be incorporated to specify domain constraints in the feature space. One can assume that domain knowledge can make the search for feature dependencies more precise; therefore, an interesting avenue for future study is to incorporate domain knowledge to complement the data-driven approaches in finding meaningful feature dependencies.

4.8 Conclusion

In this chapter, we propose a new approach to facilitate uncovering vulnerable regions within ML models employed in AMD, consequently enhancing the capability of defense mechanisms against evasion attacks, particularly realistic attacks. Specifically, we present a new interpretation of domain constraints in the feature space by extracting meaningful feature dependencies. To this end, we not only consider statistical correlations but also adopt OPF to extract such dependencies and apply them either in AE detection to identify feasible AEs or in generating feature-space RealAEs during AT to improve the robustness of detectors against RealAEs. The empirical results show the general effectiveness of our new approach in strengthening the robustness of AMD. In particular, our assessment underscores the superior efficiency and generalizability of our defense in comparison to problem-space RealAEs when it comes to adversarial hardening. Additionally, our extracted feature dependencies have proven effective in distinguishing between feasible and unfeasible ones such as unRealAEs, thereby demonstrating the significant potential for use as a reliable criterion for defenses that work based on AE detection.

4.A Evaluating the Efficacy of Learned Domain Constraints with Sparse-RS

To further assess the effectiveness of our learned domain constraints in enhancing the adversarial robustness of malware detectors, we utilize another adversarial attack known as Sparse-RS [12], which is inherently suitable for generating AEs in discrete domains like malware detection. Sparse-RS operates as a sparse adversarial attack within black-box settings, employing a heuristic search strategy—specifically, a randomized search that is effective in both discrete and continuous feature spaces. This approach not only identifies AEs with minimal changes to input features but also achieves query efficiency by reducing interactions with the target detectors.

For our evaluation, we set the initial decay factor $\alpha_{init} = 1.6$, sparsity level $k = 180$ as per [12], and a query budget $Q = 100$. Table 4.7 demonstrates that incorporating our learned domain constraints in AT to make AEs generated by Sparse-RS realizable significantly enhances adversarial robustness compared to using typical AEs generated by Sparse-RS. Moreover, our approach interestingly maintains performance on clean data better than another defense method.

4.B PiAttack

PiAttack [9], also known as PK-Greedy, is an adversarial attack that operates in the problem space and transforms Android apps into adversarial ones by attacking white-box target malware detectors. The attack adds both *primary* features to bypass malware detection and *side-effect* features to satisfy domain constraints. PiAttack consists of two main phases:

- **Initialization phase.** The first step of PiAttack is to identify the top-n benign features based on the learned weights of the linear SVM. Subsequently, for each benign feature, the attack collects a set of candidate transformations, called gadgets (i.e., slices of the apps' bytecode), by extracting them from benign apps.

- **Attack Phase.** The attack employs a greedy search strategy to find optimal perturbations. It first sorts the collected gadgets based on their feature vector's contribution to the feature vector of the malware app z, denoted by x. Next, the attack selects the best gadget from the sorted list and combines its feature vector with x. This process is repeated until x is classified as benignware. Once the perturbations have been identified, all corresponding gadgets are injected into z to generate RealAE.

Note that for some experiments that need to consider L_1 norm bound for PiAttack, we only include specific gadgets that satisfy the bound when combining their feature vector with x in the attack phase. Moreover, the study conducted in this chapter uses DNN instead of linear SVM for AT. Therefore, to clarify the significance of features, we adopt the approach suggested in [189], which utilizes the gradients provided by DNN for each feature to explain their global importance. We refer the reader to [189] for more details about the global explanation of malware detectors.

It is noteworthy that in this chapter, PiAttack has been built based on their available source codes published in [133]. Considering the recent insights from Pintor et al. [190], PiAttack seems to effectively address several key concerns highlighted in their research.

Table 4.7: The clean performance and adversarial robustness of DREBIN-DNN detectors hardened through AT using Sparse-RS, with and without incorporating our domain constraints at $\epsilon = 30$. The robust accuracy is measured against PiAttack with various attack bounds.

Defense	TPR	FPR	Clean Acc	Robust Acc		
				$\epsilon = 30$	$\epsilon = 60$	$\epsilon = 90$
AT-Unrealizable-AEs	73.9%	0.3%	95.3%	52.0%	46.2%	39.3%
AT-Realizable-AEs (ours)	80.8%	0.4%	96.3%	59.9%	50.9%	48.1%

- **Optimization Challenges and Loss Saturation.** [190] highlights the issue of loss saturation, where further modifications fail to improve attack success. While PiAttack uses a greedy search rather than gradient-based optimization, it can similarly get stuck in suboptimal solutions. Nevertheless, PiAttack reduces this risk by meticulously choosing gadgets during the initialization phase based on their potential to shift the classification score toward the benign class. This approach ensures the optimization process remains effective, preventing the stagnation that might occur with a less structured search.

- **Feature Contribution and Impact.** PiAttack effectively manages the contribution of each feature. During the initialization phase, gadgets are carefully pre-selected based on their ability to positively influence the classification outcome while avoiding those that might unintentionally reinforce a malicious label. This strategy prevents issues similar to gradient vanishing or exploding, ensuring that each modification significantly contributes to the attack's success without compromising the app's benign appearance.

- **Maintaining Problem-Space Feasibility.** PiAttack is crafted to maintain the original functionality of the Android app. This is accomplished by encapsulating the injected code in conditional statements that are never executed at runtime, ensuring the app's original behavior remains intact. Additionally, PiAttack uses opaque predicates to guard against static analysis tools that might otherwise eliminate the transplanted code. These strategies ensure that the generated adversarial examples remain functional and plausible within the problem space.

4.C PK-Feature

This is a white-box attack that iteratively perturbs the impactful features of a malware sample until it reaches the maximum allowable perturbation. It's essential to note that the impactful features are those that exert a more significant influence on the classification outcomes compared to other features, and like PiAttack, they are determined based on the weight parameters of the linear SVM learned during training. Incorporating the dependent features of each perturbed feature, based on our extracted meaningful feature dependencies, enables this attack to generate feature-space RealAEs.

4.D AT with Non-Uniform Perturbations

Unlike conventional feature-space adversarial attacks used in AT that can only perturb the training sample under norm-bounded constraints, the main idea of AT with non-uniform perturbations [161] is to take into account the data distribution of training data by allowing attackers to generate non-uniform perturbations. Specifically, the paper proposes a new projection approach for the PGD attack as follows:

$$P(\Omega\delta) = \begin{cases} \epsilon \frac{\delta}{\|\Omega\delta\|_p} & \text{if } \|\Omega\delta\|_p > \epsilon \\ \delta & otherwise \end{cases} \tag{4.6}$$

where $\|.\|_p$ shows the L_p norm bound, $\Omega \in \mathbb{R}^{d \times d}$ is a diagonal matrix that can be specified by a *weighted norm*, and d is the dimensions of samples in the training set. By incorporating Ω in the projection, PGD can perturb important features more than less important features that may have a lesser effect on classification. One of the suggestions of the paper to capture the importance of features is to utilize *Pearson's correlation coefficient* of each feature f_j with the label y, which is denoted as $|p_{j,y}|$. Note that Ω is constructed using this coefficient as follows:

$$\Omega = \frac{diag(\{p_{j,y}^{-1}\}_{j=1}^{d})}{\|diag(\{p_{j,y}^{-1}\}_{j=1}^{d})\|_2} \tag{4.7}$$

where $p_{j,y}^{-1} = 1/p_{j,y}$.

5

Enhancing Adversarial Robustness with Robust Feature Space

Machine learning (ML) has demonstrated significant advancements in Android malware detection (AMD); however, the resilience of ML against realistic evasion attacks remains a major obstacle for AMD. One of the primary factors contributing to this challenge is the scarcity of reliable generalizations. Malware classifiers with limited generalizability tend to overfit spurious correlations derived from biased features. Consequently, adversarial examples (AEs), generated by evasion attacks, can modify these features to evade detection. This chapter proposes a domain adaptation technique to improve the generalizability of AMD by aligning the distribution of malware samples and AEs. Specifically, we utilize meaningful feature dependencies, reflecting domain constraints in the feature space, to establish a robust feature space. Training on the proposed robust feature space enables malware classifiers to learn from predefined patterns associated with app functionality rather than from individual features. This approach helps mitigate spurious correlations inherent in the initial feature space. Our experiments conducted on DREBIN, a renowned Android malware detector, demonstrate that our approach surpasses the state-of-the-art defense, Sec-SVM, when facing realistic evasion attacks. In particular, our defense can improve adversarial robustness by up to 55% against realistic evasion attacks compared to Sec-SVM.

5.1 Introduction

Despite the substantial progress in utilizing machine learning (ML) for Android malware detection (AMD), the field still suffers from security concerns surrounding ML models, especially their vulnerabilities to realistic evasion attacks. These attacks change malware apps into adversarial examples (AEs), tricking AMD while preserving the malware's properties, e.g., their executability and malicious functionalities. Evasion attacks can circumvent ML models, exploiting their susceptibility to learning vulnerabilities [8], which often arise from

This chapter is based on the published paper: H. Bostani, Z. Zhao, and V. Moonsamy, Improving Adversarial Robustness in Android Malware Detection by Reducing the Impact of Spurious Correlations, 29th European Symposium on Research in Computer Security International Workshops (ESORICS 2024 International Workshops), 2024 [191]. The content remains unchanged from the published version.

the limited generalizability inherent in ML models. In fact, adversaries deceive ML models by generating AEs that sufficiently deviate from the distribution of the training samples [192].

One of the major factors contributing to the generalizability challenges of ML is *biased features*[193]. These features introduce biases into the model, potentially resulting in poor performance on unseen samples. ML models tend to learn these simple cues effectively on most training samples, which in turn causes them to perform well on those samples; however, they encounter difficulty with more complex unseen samples (e.g., AEs) due to the distribution shift [193]. Specifically, the presence of biased features causes classifiers to learn *spurious correlations* [194], resulting in misleading associations between biased features and the target variable, known as the *label* in supervised learning. In other words, ML models might learn misleading patterns—the irrelevant associations between features and the label—that may not generalize well to unseen samples with distributions different from those of the training samples. These misleading correlations often occur because the training set fails to accurately represent the true data distribution, typically due to sampling bias [195]. Spurious correlations, substantially diminish the generalizability of ML models, especially in the context of cybersecurity. These meaningless correlations represent patterns within the data unrelated to the security problem but serve as shortcuts for distinguishing classes [48]. For instance, the inclusion of particular market information, such as Chinese markets, in numerous malware samples might cause the ML model to mistakenly associate this feature with maliciousness [48], instead of prioritizing the identification of authentic patterns linked to malicious behavior.

Spurious correlations present intriguing implications for realistic evasion attacks. Adversaries aim to append adversarial payloads that significantly influence features crucial for classification while ensuring that the added contents are unrelated to the app's functionality. Therefore, realistic evasion attacks could leverage the features associated with spurious correlations, as they influence the classification outcome while ensuring the malicious patterns remain intact. Since learning spurious correlations poses challenges for malware classifiers when the distribution of AEs significantly differs from that of malware samples in the training set, this issue can be reduced if both malware samples and AEs follow a similar distribution. This chapter introduces a novel domain adaptation[1] technique designed to reduce the impact of spurious correlations by aligning the distributions of the source domain (including malware samples) and the target domain (including AEs). As illustrated in Figure 5.1, to reduce the adverse impact of spurious correlations on the adversarial robustness of AMD, our proposed approach utilizes domain constraints that characterize app properties to create a robust feature space. To this end, we first model domain constraints based on the relationships between features derived from the feature representations of the training apps. Within the feature space, domain constraints denote complex relationships among features that an adversary must fulfill for an attack to be realistic [167]. Then, we propose a transformation function using these identified patterns to transform samples from the initial feature space to a robust feature space. The distribution of malware apps is expected to align more closely with adversarial ones when represented in the robust feature space rather than in the initial feature space. This is because, unlike the features in the initial feature space, each feature in the robust space reflects a functional aspect (e.g., sending an SMS) commonly shared among feasible apps, including both malware and adversarial apps. Our

[1] In domain adaptation [196], *domain* refers to a certain distribution over a sample set (e.g., the training set).

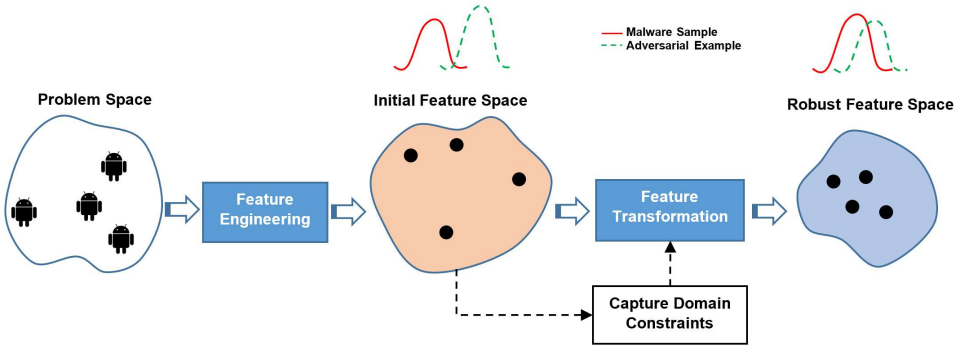


Figure 5.1: An illustration of our proposed domain-adaptation technique. In the initial feature space, the distributions of malware samples and adversarial examples differ significantly. However, in our proposed robust feature space, their distributions are more aligned.

contributions[2] can be summarized as follows:

- We propose a robust feature space based on a novel domain adaptation approach to reduce spurious correlations, thereby enhancing the adversarial robustness of AMD against realistic evasion attacks.
- We empirically demonstrate that our proposed defense surpasses the state-of-the-art defense, Sec-SVM [16], in hardening AMD against gradient-based and query-based realistic attacks across various threat models.
- Our empirical findings illustrate that the distribution of malware apps and AEs is more aligned in the proposed robust feature space than in the initial feature space.

The rest of the chapter is organized as follows: Section 5.2 provides an elaboration on the fundamental concepts crucial to the chapter. Our novel approach for constructing a robust feature space is detailed in Section 5.3. Section 5.4 assesses the effectiveness of our proposed defense technique in hardening the robustness of ML-based AMD against realistic evasion attacks. Section 5.5 examines relevant studies that have explored feature representations to enhance the adversarial robustness of malware detection. The limitations and potential directions for future research are discussed in Section 5.6 and we conclude in Section 5.7.

5.2 Background

In this section, we briefly review the fundamentals of evasion attacks, spurious correlations, and domain constraints.

5.2.1 Evasion Attacks

Consider $\phi : Z \rightarrow X$ as a mapping function representing Android apps of the problem space Z with d-dimensional feature vectors in the feature space X. ML-based AMD is a

[2]We make our code publicly available at `https://github.com/HamidBostani2021/robust-feature-space` to allow reproducibility.

binary classifier $f : X \rightarrow Y$ equipped with a discriminant function $g : X \times Y \rightarrow \mathbb{R}$. Here, $f(x) = \arg\max_{i \in Y} g_i(x)$ assigns labels to $x \in X$, where $Y = \{0, 1\}$ denotes the label space with $y = 0$ representing benign labels and $y = 1$ representing malicious labels. In the binary feature space [29, 143, 197], each element of the feature vector $x \in X$ is binary, where 0 signifies absence and 1 signifies the presence of specific features. It is noteworthy that $F = \{f_1, f_2, ..., f_d\}$ represents the feature set defining the dimensions of X, where d denotes the number of dimensions.

Generally, evasion attacks generate AEs by altering $x \in X$ through the discovery of optimal perturbations δ applied to it. Particularly, malicious actors endeavor to solve the following optimization problem [9, 16]:

$$\arg\max_{\delta} \quad g_0(x' = x + \delta) \quad \text{subject to} \quad \delta \models \Omega, \tag{5.1}$$

where the perturbation vector δ must satisfy constraints defined in the feature space denoted by Ω, such as a naive norm bound [9].

5

5.2.2 Spurious Correlations

In statistical analysis, spurious correlation describes a scenario in which two variables seem to be associated, but their relationship is either accidental or influenced by an external factor [194]. Such situations can result in misleading or erroneous interpretations of data and models [194]. Supervised learning algorithms are vulnerable to spurious correlations because classifiers often tend to learn any signal in the dataset that maximizes accuracy, even those that may appear incomprehensible to humans [198]. Spurious correlations pose significant challenges for ML in cybersecurity because existing ones may lead ML models to learn patterns in the data unrelated to the security problem, thereby creating shortcuts for classifying classes [48]. Over the past few years, several approaches (e.g., invariant learning [199] and group robustness [200]) have been proposed to mitigate the impact of spurious correlations in ML. Domain adaptation stands out as one of the approaches aimed at aligning the distribution between source and target domains to address spurious correlations. This approach focuses on transferring knowledge learned from the source domain to the target domain, thereby aiding the model in better generalizing to new data and reducing its dependence on spurious correlations [201]. In the study conducted in this chapter, we concentrate on domain adaptation by suggesting a robust feature space, as a high-quality feature representation is vital for the success of domain adaptation [196].

5.2.3 Domain Constraints in the Feature Space

Generally, evasion attacks must account for domain constraints to generate realizable AEs. In the problem space, the domain constraints consist of available transformations, preserved semantics, robustness to preprocessing, and plausibility, as formalized by Pierazzi et al. [9]. These constraints signify the properties of malware apps that must be maintained after manipulation in the problem space. However, domain constraints appear differently in the feature space. During the past years, numerous studies in diverse contexts have demonstrated that feature dependencies serve as indicators of domain constraints within the feature space [151, 152, 167–169]. The study conducted in this chapter applies the idea proposed by [152] to infer domain constraints based on feature dependencies. This approach suggests

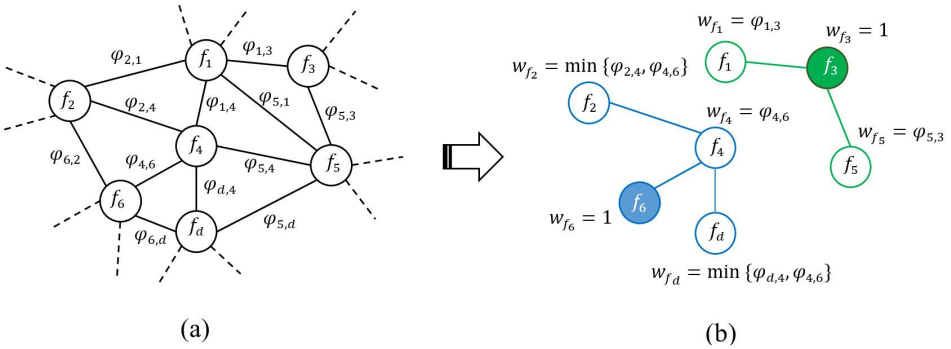


Figure 5.2: An example of the OPF process constructed to capture feature dependencies. (a) shows the completed weighted graph G, where node f_i represents the feature f_i and $\varphi_{i,j}$ represents the correlation between f_i and f_j. (b) the final OPF comprises two OPTs derived from G. The colored nodes signify primary features, and w_{f_i} denotes the path cost from f_i to its relevant primary features.

that the relationship between primary and secondary features can model domain constraints. Primary features are those that limit the range of permissible values for other features, whereas secondary features do not impose any such limitations. According to [152], primary features can be identified based on data observations by indicating the features that are most correlated with others. Here, we not only employ correlation to specify the primary features but also adapt Optimum-path Forest (OPF), as used in [137], to determine the secondary features. Using the primary features as a starting point, we apply OPF to partition the remaining features into different groups, ensuring that each cluster includes relevant features. (i.e., primary and secondary features). In fact, we partition F, the feature set characterizing X, into distinct clusters denoted as A_i. This clustering strategy aims to identify primary and secondary features by grouping them together. As shown in Figure 5.2, the technique involves constructing a complete weighted graph, denoted as $G = (V, E)$, where $G = F$ and $E = F \times F$ encompassing edges that connect each feature pair (f_a, f_b), with weights determined by a correlation coefficient. Following the graph construction, the next step involves partitioning G into various clusters such as A_i where it constitutes a subset of all features within the feature space, i.e., $A_i \subset F$. It is noted that the OPF algorithm treats each cluster as an Optimum-path Tree (OPT), wherein each feature has an optimal path to the OPT's prototype, which is a primary feature in our case. The path cost is the minimum weight of the edges along the path. For more details about primary and secondary features, refer to [152], and for information on OPF, see [137].

5.3 Our Proposed Defense

This section illustrates how domain constraints, specified by feature dependencies, have been utilized to propose a robust feature space. The proposed defense aims to enhance the robustness of ML-based AMD against realistic attacks by reducing the impact of spurious correlations. To this end, we first formulate a new domain adaptation approach, which leverages the feature dependencies of the source domain to build a resilient feature space. Subsequently, we delve into an elaborate discussion demonstrating the process of constructing

our proposed robust feature space.

5.3.1 Formulation of the Problem

Suppose X denotes the initial feature space, and T denotes the source domain (i.e., training set), and U implies the target domain (i.e., unseen samples). Moreover, let Y, ${D_T}^X$, and ${D_U}^X$ represent the label space and the data distributions of T and U based on X, respectively. Given a labeled source dataset $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$ drawn from ${D_T}^X$, the goal is to build a robust feature space H that captures the dependencies observed in X and can be effectively used for classification in both T and U. We can mathematically express this as follows:

1. Construct a feature transformation function $\lambda : X \rightarrow H$ that maps the initial feature space X to the robust feature space H. This function is designed to capture the relevant feature dependencies observed in X considering the source domain and build a new feature representation based on them. Indeed, the transformation function tends to construct H wherein ${D_T}^H$ and ${D_U}^H$ are more align compared to ${D_T}^X$ and ${D_U}^X$.

2. Train a classifier $f : H \rightarrow Y$ using the transformed features H. This classifier should be capable of making accurate predictions based on robust feature representations. The objective can be formulated as minimizing a loss function J over the labeled source dataset:

$$\min_{\lambda, f} \sum_{i=1}^{n} J(f(\lambda(\mathbf{x}_i)), y_i)$$

where J represents the classification loss function, and $f(\lambda(\mathbf{x}_i))$ denotes the predicted label for the i-th source domain sample after feature transformation.

5.3.2 Robust Feature Space

The primary objective of our proposed robust feature space is to mitigate the impact of adversarial perturbations on misclassification by aligning the distributions of malware samples seen during training and AEs. Leveraging the proposed robust feature space adapts the ML-based malware detection, trained on training samples, to perform well on samples with unseen distribution, especially AEs. Generally, the adversarial perturbations used in realistic attacks consist of redundant codes that are unrelated to the malicious functionality of the original malware apps. Therefore, we anticipate that their adversarial effect can be diminished if our ML model can learn authentic malicious patterns rather than shortcut patterns (i.e., spurious correlations [48]) unrelated to malware detection. To achieve this, we propose a new transformation function (i.e., $\lambda : X \rightarrow H$) that maps each $x \in X$ to an $h \in H$ based on feature-space domain constraints. ML used in AMD operates with feature representations of apps in H, which implicitly abstract domain constraints, instead of X in both the training and inference phases. We expect that training our ML-based detector on H gives the detector more chances to learn generic attack patterns to distinguish malicious and benign apps because H is characterized based on the pre-identified patterns (i.e., a collection of feature groups, where each includes the interdependent features) that represent domain constraints. Indeed, training an ML model on H will cause the detector to rely on groups of features in X rather than individual features, potentially introducing bias in classification.

Suppose $\Lambda = \{A_1, A_2, ..., A_m\}$ represents all feature dependency clusters identified by utilizing the method discussed in Section 5.2.3, where $A_i \subset F$ denotes the feature set of

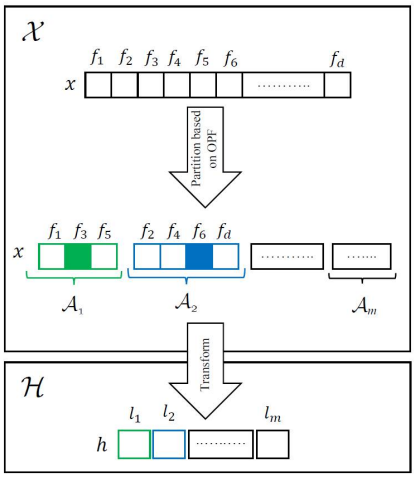


Figure 5.3: Overview of our method for applying domain constraints to construct a robust feature space.

5

the i-th cluster in Λ. Additionally, let L denote the dimensions (i.e., features) of the feature space H. As depicted in Figure 5.3, within the proposed new feature space H, each A_i is associated with a feature $l_i \in L$ serving as the representative of all features in A_i to ensure the detection model's accuracy on legitimate samples. This representativeness is necessary since A_i encompasses interdependent features with similar information about the target class, making a single feature in L sufficient to represent these relevant features. To determine whether a feature $l_i \in L$ should appear in H for a sample $x \in X$, we utilize an activation function based on the sigmoid. This function can transfer the influence of input features in A_i to the output feature l_i while uniformly increasing the probability of a feature appearing in the output as the number of input features rises. Furthermore, the function ensures that changing a feature $f_j \in A_i$ cannot simply alter l_i due to our aim to increase the evasion costs for adversaries. It is important to note that the sigmoid function, being a monotonic function, adjusts l_i based on the features in A_i. As the sigmoid function exhibits an *S-Shaped Curve* [202], we anticipate that it can help smooth out the severe impact of adversarial perturbations within A_i on l_i. This is because, when there are large adversarial fluctuations or spikes in the input feature values in A_i, the output will show a more gradual change due to the sigmoid's property of saturating large values. According to the proposed transformation method, the value of feature l_i in $h = \lambda(x)$ is computed as follows:

$$l_i = \begin{cases} 1 & \sigma(s = \sum_{\forall f_j \in A_i} w_j \cdot x_j) > \theta \\ 0 & \text{otherwise,} \end{cases} \tag{5.2}$$

where $\sigma(\cdot) = \frac{1}{1+e^{-s}}$ represents the sigmoid function, w_j stands for the weight of feature f_j in A_i, x_j denotes the value of feature f_j in x, and θ signifies a threshold for activating l_i in H. It's important to note that not every feature holds the same level of importance (e.g., some features may arise due to noise). Thus, we aim to account for the greater impact of features that might contribute more to the detection task. Consequently, as shown in equation (5.2),

we take into consideration the weights of features since they can indicate the importance of features in A_i. In the proposed transformation function, we consider the path cost of each feature in the constructed OPF as the weight for the features, since it serves as a measure demonstrating the relevance of a certain feature to its respective cluster. Moreover, in equation (5.2), $s \geq 0$ since w_j possesses a positive value. Hence, $0.5 \leq \sigma \leq 1$ in our context, implying that θ should be chosen from the interval $(0.5, 1)$. Generally, opting for a moderate threshold seems preferable because setting a very low threshold may result in the output feature appearing even with minor perturbations in the input features. This occurs because triggering more features in A_i leads to a larger σ, potentially causing $l_i = 1$ if θ is small. On the other hand, a very high threshold hinders the transfer of the input features' effect to the output. In essence, although a high threshold might enhance the model's robustness by reducing the impact of adversarial perturbations on features in A_i, it could decrease the model's accuracy on legitimate samples.

5.4 Experiments

In this section, we conduct empirical assessments to gauge the effectiveness of the proposed defense mechanism against various realistic evasion attacks. All experiments were conducted on a Debian Linux workstation equipped with an Intel(R) Core(TM) i7-4770K CPU running at 3.50 GHz and 32 GB of RAM.

5.4.1 Experimental Setup

Dataset. The study conducted in this chapter utilizes an available dataset [9] comprising approximately $170K$ Android apps sourced from AndroZoo [125]. An app within this dataset is classified as benign if no VirusTotal Antiviruses (AVs) detect it, while it is deemed malware if it is flagged by four or more AVs. Our training set comprises $50K$ randomly selected samples, with $30K$ samples allocated for the test set for evaluating Android malware detectors. The training set consists of $45K$ clean samples and $5K$ malware samples, whereas the test set comprises $25K$ clean samples and $5K$ malware samples. All samples are encoded based on the DREBIN [96] feature space before being processed by the malware detectors. Given that DREBIN encompasses a vast but sparse feature space, we select the $10K$ most frequently occurring features, as recommended by prior research [16, 25]. To evaluate the adversarial robustness of various Android malware detectors, we employ $1K$ malware samples as outlined in [84] to generate AEs.

Threat Models and Attacks. Adversarial attacks can be analyzed based on their objectives, knowledge, and capabilities. The adversary's objective is to cause misclassification in AMD, resulting in the classification of the adversarial (malware) examples as benign ones. Moreover, the adversary's knowledge of the target model, including its training data, feature space, and parameters, ranges from perfect (PK) to limited (LK) or zero (ZK). In PK, LK, and ZK attacks, the target model is perceived as a white-box, gray-box, and black-box model, respectively. Finally, the adversary's capability enables the generation of AEs either within the feature space, by altering feature representations of Android malware apps, or within the problem space, through a sequence of transformations applied to Android malware apps [9].

The study conducted in this chapter explores two realistic problem-space attacks, namely *PK-Greedy* [9] and *EvadeDroid* [84], to evaluate the adversarial robustness of malware

detectors discussed in Section 5.4.2. PK-Greedy and EvadeDroid transform Android apps into adversarial instances by targeting white-box and black-box malware detectors, respectively. The details of these attacks are described as follows.

- **PK-Greedy** [9] generates problem-space realizable AEs by applying effective transformations (i.e., code snippets called gadgets extracted from donor apps) specified by feature-space perturbations on the target model. This attack adds not only *primary* features to bypass malware detection but also *side-effect* features to meet the domain constraints. The attack was originally tested in the PK setting, but here we also test it in an LK setting where the AEs transfer from a surrogate model to a target model. To utilize PK-Greedy in PK settings when the feature space is H, we must adapt this attack to target the ML model trained on the robust feature space, as it was initially designed for models trained on the DREBIN feature space. Therefore, PK-Greedy is an adaptive one aware of our proposed transformation function. Specifically, during the attacking phase, PK-Greedy identifies the most adversarially sensitive features in L based on the model trained on H where L is the feature set characterizing H. Then, for each identified feature $l_i \in L$, it finds a transformation wherein its triggered features (i.e., the DREBIN features that can appear in an app after applying the transformation) have a significant overlap with the features in A_i (i.e., the cluster in the DREBIN feature set corresponding to $l_1 \in L$), and then applies it to the app.

- **EvadeDroid** [84] generates problem-space realizable AEs through a sequence of transformations by querying the target model in a ZK setting. This adversarial attack involves the initial collection of problem-space transformations by extracting code snippets containing API calls from benign apps found in the wild, resembling malware apps. Subsequently, random search is employed to select and apply transformations that induce the malware app to exhibit similarities to benign apps. In addition to the original ZK setting, here we also consider a more restricted setting where EvadeDroid is only allowed to query a surrogate model and then transfer the AEs to the target model. Specifically, we set the query budget $Q = 10$, and $\alpha = 50\%$ (i.e., the percentage of the relative increase in the size of a malware sample after manipulation).

Evaluation Metrics. To assess the malware detectors, we test both their clean performance and robustness. For clean performance, we compute Clean Accuracy, True Positive Rate (TPR), and False Positive Rate (FPR) on benign and malware samples. For robustness, we compute Robust Accuracy on adversarial malware examples. Additionally, we include the average number of added features needed to achieve successful AEs.

5.4.2 Evaluation of Proposed Defense

This section aims to evaluate our robust feature space introduced in Section 5.3. We consider the following four ML-based Android malware detectors:

- **DREBIN-Original** [96], a well-known Android malware detector that is based on the linear Support Vector Machine (SVM). It is trained with the original DREBIN feature space.

- **Sec-SVM** [16] is the secure version of DREBIN-Original for strengthening the robustness of linear SVM against AEs. Sec-SVM relies on more features, and this

Table 5.1: The training time (TR), clean performance metrics—including TPR, FPR, and Clean Acc—and robust accuracy—including R_Acc_P (robust accuracy against PK-Greedy) and R_Acc_E (robust accuracy against EvadeDroid)—are reported for various DREBIN detectors. Realizable AEs are transferred from DREBIN-Original. "*" means both the surrogate and target models are DREBIN-Original.

Model	TR	TPR	FPR	Clean Acc	R_Acc_P	R_Acc_E
DREBIN-Original	8.2s	87.2%	1.4%	96.7%	*0.0%	*26.8%
Sec-SVM	25.4s	77.0%	1.0%	95.3%	97.6%	72.1%
DREBIN-FeatureSelect	1.3s	79.5%	1.3%	95.5%	30.7%	49.8%
DREBIN-Robust (ours)	1.7s	77.5%	1.3%	95.1%	97.9%	94.6%

increases the evasion cost. Essentially, the goal of Sec-SVM is to enhance the robustness of a linear SVM by ensuring that it assigns weight more evenly across all features used in the model. This approach inherently makes generating AEs more challenging for an attacker, as it needs to alter more features to bypass malware detection.

- **DREBIN-Robust** is our robust DREBIN detector trained with our new robust feature space.

- **DREBIN-FeatureSelect** resembles DREBIN-Original but is trained with a lower-dimensional feature space to enhance the stability of models against noise [203]. Feature selection aims to eliminate redundant or irrelevant features, which can be misused by adversarial perturbations, and thus improve the robustness of an ML model. We utilize Linear SVC to identify the 500 most influential features, resulting in clean accuracy comparable to DREBIN-Robust.

In the first experiment, we assess the robustness of malware detectors against transferable AEs generated by PK-Greedy and EvadeDroid, using DREBIN-Original as the surrogate model. It's ensured that the AEs are generated from malware samples correctly detected by

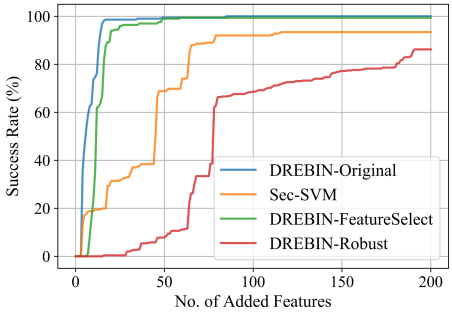


Figure 5.4: The evasion success rates of PK-Greedy against different DREBIN detectors when varying the number of added features.

Table 5.2: The robustness of different DREBIN detectors against realizable AEs that are directly generated on the corresponding models in terms of Robust Acc and Number of Features (NoF).

Model	PK-Greedy		EvadeDroid	
	Robust Acc	NoF	Robust Acc	NoF
DREBIN-Original	0.0%	9.2	26.8%	66.5
Sec-SVM	6.6%	37.9	31.9%	77.5
DREBIN-FeatureSelect	0.6%	19.6	45.9%	62.1
DREBIN-Robust (ours)	13.8%	86.1	87.0%	56.7

all four malware detectors, and the results are computed based on the successful AEs for DREBIN-Original. As depicted in Table 5.1, all defenses achieve similar clean performance in terms of TPR, FPR, and clean accuracy; however, the training time of our proposed defense is substantially shorter than that of Sec-SVM and even DREBIN-Original. It is important to note that, despite the significant improvement in training time, our technique incurs some overhead due to the creation of the robust feature space and the transformation of training samples into this space. However, this overhead is minimal compared to the feature engineering required during preprocessing to extract and represent features from apps in the initial feature space. Concerning robustness, although DREBIN-FeatureSelect demonstrates notable robustness compared to DREBIN-Original due to feature selection, both DREBIN-Robust and Sec-SVM exhibit significantly higher robustness. Additionally, our DREBIN-Robust outperforms Sec-SVM, especially for EvadeDroid.

We further examine a more challenging scenario where AEs are directly generated on the target model. Due to the time-consuming nature of generating problem-space AEs across different detectors, we limit our test to 500 malware samples. Moreover, to ensure rigorous evaluation of detectors in worst-case scenarios, PK-Greedy operates in the PK setting, especially in attacking the ML model trained on the robust feature space (i.e., DREBIN-Robust). As illustrated in Table 5.2, the superiority of our DREBIN-Robust over Sec-SVM remains evident, particularly in defense against EvadeDroid. Indeed, DREBIN-Robust demonstrates superior performance compared to Sec-SVM against both attacks, with a 7.2% improvement against PK-Greedy and a 55.1% improvement against EvadeDroid. Additionally, in defense against PK-Greedy, DREBIN-Robust escalates the evasion cost by necessitating the adversary to modify significantly more features to achieve success. Figure 5.4 further validates the consistently superior performance of our DREBIN-Robust across varying numbers of added features. Furthermore, we observe rapid convergence of the evasion rates, suggesting that increasing the number of added features scarcely improves PK-Greedy against our DREBIN-Robust. It should be noted that although our defense in DREBIN-Robust substantially improves robustness against both realistic attacks, the success rate of PK-Greedy is significantly higher than that of EvadeDroid—specifically, 73.2%—because PK-Greedy is an adaptive attack that operates in PK settings, whereas EvadeDroid targets our proposed detector in ZK settings.

Note that in Table 5.2, the fact that the average number of added features required by EvadeDroid for bypassing DREBIN-Robust is lower than bypassing the other detectors is due to the property of EvadeDroid. Specifically, in EvadeDroid, a transformation is applied to a malware app only if it can increase the chance of generating successful AEs. This leads to

the difference between the number of transformations applied to the detectors (e.g., 1.00 for DREBIN-Robust vs. 2.54 for Sec-SVM). This difference indicates that most transformations are not good enough for attacking DREBIN-Robust, and consequently leads to a difference in the number of added features.

5.4.3 Discussion

Our empirical investigation highlights the resilience of the proposed robust feature space against realizable AEs generated by various realistic evasion attacks. To further analyze this observation, Figure 5.5 displays the t-SNE visualization of malware samples from the training set and AEs from the test set. The visualization is based on the top-100 important features selected using Linear SVC, both in the original feature space and our proposed robust feature space. It is noteworthy that AEs are generated by targeting DREBIN-Original with PK-Greedy. In the proposed robust feature space, the visualization demonstrates a closer alignment between the distribution of malware samples and AEs compared to those observed in the initial feature space. Consequently, this contributes to the enhanced adversarial robustness observed in DREBIN-Robust trained on H compared to DREBIN-Original trained on X, as demonstrated in the results presented in Table 5.1.

Moreover, to grasp the extent of our approach's efficacy in countering potentially misleading correlations learned by models, we delve into the operational mechanism of DREBIN, (i.e., a linear SVM-based detector). The detection model relies on a score function, derived from the inner product of the model's parameters (i.e., learned weights $\vec{W}$) and a feature vector representing an app z. Specifically, this function is denoted as $f(z) = \langle \phi(z), \vec{W} \rangle$ when the model is trained on X, and as $f(z) = \langle \lambda(\phi(z)), \vec{W} \rangle$ when trained on H. A sample is classified as benign if $f < 0$, and conversely if $f > 0$. This suggests that features with high negative weights play a pivotal role in classifying a sample as benign. Adversaries target altering the features relevant to negative weights as it increases the chance

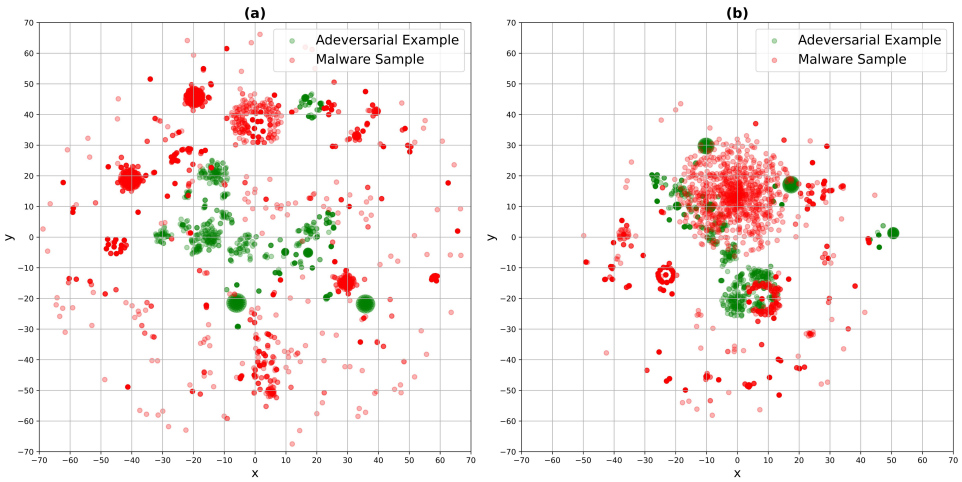


Figure 5.5: t-SNE visualization of malware and adversarial malware samples in (a) the feature space X and (b) our robust feature space H.

```
{
    "f1": [
      'URLs::https://play.google.com/store/apps/'
    ],

    "l1": [
      'Activities::xmlparser.GiftActivity',
      'URLs::http://jgpre.alibaba.inc.com/',
      'URLs::http://jg.daily.taobao.net/',
      'URLs::http://jg.alibaba.inc.com/',
      'Activities::xmlparser.SplashScreenActivity',
      'Activities::xmlparser.PrivacyActivity',
      'S_and_P::android.permission.BIND_REMOTEVIEWS'
    ]
}
```

Figure 5.6: The details of features $f_1 \in X$ and $l_1 \in H$.

of deceiving malware detectors. The features that are important for the classifier operating on X might be biased features; however, H aims to diminish classifier bias by assisting it in relying on sets of features that contribute more to the behavior of the apps, rather than on individual features. For instance, as shown in Figure 5.6, $f_1 \in X$, the feature with the most negative weight in the linear SVM trained on X, can potentially represent a shortcut feature that might be important due to biased data, while $l_1 \in H$, the feature with the most negative weight in the linear SVM trained on H, seems relevant to functionality.

We conduct a further evaluation to ascertain whether DREBIN-Robust exhibits superior resilience compared to DREBIN-Original in learning spurious correlations. To ensure bias in a feature, we must identify a feature that not only seems biased but also is absent in some of the malware samples in our test set, allowing us to observe the effects of adding it to the malware samples. Note that a feature appears biased when it is prevalent in the majority of benign samples but is present in the minority of malware samples within the training set. Among features with negative weights, feature f_{44} (i.e., `android.permission.INTERNET`) stands out as the first one that not only appears biased but is also absent in some of the malware samples in the test set. Adding this feature to malware samples of the test set that lack f_{44}, drops DREBIN-Original's robustness from 96.4% to 86.3%. This demonstrates f_{44}'s bias, resulting in spurious correlations in DREBIN-Original. This is because the feature is not effective in distinguishing between benign and malware samples since it can potentially be present in both types of samples. However, DREBIN-Robust remains unaffected by this misleading correlation, demonstrating its resilience against modifications to feature f_{44}.

Ethical Considerations. Since our proposed defense strategy is designed to enhance cybersecurity and mitigate adversarial attacks, rather than facilitate malicious activities, ethical concerns are minimal. However, we stress that our defense mechanisms should be used responsibly and primarily as a baseline for research purposes.

5.5 Related Work

Despite numerous efforts aimed at enhancing the adversarial robustness of ML-based AMD against evasion attacks (e.g., [14, 17, 24, 25, 28, 154]), few studies have primarily explored the impact of features on enhancing adversarial robustness. To this end, Demontis et al. [16] introduced Sec-SVM which trains a linear SVM with a more uniform distribution of feature weights. This study ensured that the linear model learned feature weights more evenly by

applying box constraints on weights within the standard optimization problem used in the linear SVM. Their proposed Sec-SVM relies on a larger number of features for classification, thereby enhancing adversarial robustness, as attackers would need to perform significantly more meticulous manipulations to generate AEs. Chen et al. [15] introduced SecureDroid, a defense strategy employing ensemble learning, coupled with a novel feature selection technique, to bolster classifier resilience against evasion attacks. The proposed method highlighted the importance of individual features, considering both their contribution to classification and their vulnerability to manipulation by attackers. In other words, the paper argues that the features with greater significance in classification and lower manipulation cost are interesting features for attackers. Specifically, the proposed method reduced the presence of these features by altering the training set. This resulted in a more uniform distribution of feature importance, compelling attackers to manipulate a larger array of features to bypass detection. Yang et al. [28] investigated weight bounding, similar to [16]. They constrained the weights of the linear classifier on a few dominant features to achieve more evenly distributed feature weights. They determined the dominant features by noting that adversaries could generate evasive malware variants with minimal mutations on certain features identified as dominant, compared to the extensive mutations required on other non-dominant features. Chen et al. [18] introduced a gradient masking method that converts the binary feature space into continuous probabilities, encoding the distribution for both benign and malicious instances. The paper argues that when the binary feature space is transformed into a continuous space, the gradient of feature addition or removal accessible to attackers may be significantly reduced. As a result, attackers cannot easily bypass detection. This technique also enables the classifier to strike an optimal balance between security and accuracy by utilizing a softmax function with an adversarial parameter.

5.6 Limitations and Future Work

While our experiments on DREBIN convincingly showcase the effectiveness of our proposed defense against realistic evasion attacks, it remains imperative to extend our evaluation to encompass a broader array of malware detection systems. By doing so, we can ascertain the generalizability of our approach across different detectors. Furthermore, since the technique for capturing domain constraints is data-driven, it is essential to periodically update the feature space before regular retraining to keep ML models effective. This is crucial for maintaining the performance of malware classifiers, especially against evolving zero-day malware with varying feature dependencies. In addition, as elucidated in Section 5.3.2, the selection of an appropriate threshold is pivotal for ensuring the efficacy of our defense. Hence, future inquiries should delve into the impact of different threshold values on both the clean and robust accuracies of our defense mechanism.

5.7 Conclusion

This chapter introduces a novel defense mechanism based on domain adaptation to enhance the adversarial robustness of ML-based AMD. The proposed method aims to enhance the reliable generalizability of AMD against adversarial examples by mitigating spurious correlations misused by evasion attacks. Our approach leverages domain constraints to establish a robust feature space, enabling ML models to learn genuine malicious patterns

of Android malware. Experimental results on DREBIN, a well-known AMD, demonstrate significant improvements over the state-of-the-art defense Sec-SVM, particularly against realistic evasion attacks.

6

Enhancing Adversarial Robustness with Robust Optimization

Adversarial Training (AT) has emerged as a promising defense against adversarial evasive attacks, yet its ability to effectively harden malware classifiers without sacrificing the accuracy on clean data remains an intricate challenge. Assessments of AT often depend on weak or unrealistic evasion attack scenarios, failing to reveal the practical challenges posed by real-world adversarial threats. Prior work treats robustness as a task-dependent property, often focusing narrowly on aspects like adversarial confidence or adversarial example realism. This chapter challenges these assumptions and proposes a novel framework to systematically evaluate AT's effectiveness in malware classification. Our framework tackles this complex problem by breaking it down into multiple key dimensions—whose behaviors remain largely unknown—and empirically examining the interconnected roles of diverse factors across them. These dimensions include data, feature representations, classifiers, and robust optimization settings, analyzed in our framework through reliable evaluation practices, such as realistic evasion attacks. By adopting this holistic approach, we thoroughly evaluate AT properties within the malware domain. This enables deeper insights into how these factors collectively influence both clean and robust accuracy, providing fresh perspectives that challenge existing studies. Our findings reveal five critical evaluation pitfalls that affect state-of-the-art research. We also summarize our insights into ten takeaways, along with practical recommendations to guide future research toward uncovering the conditions under which AT achieves optimal performance.

6.1 Introduction

Although Machine Learning (ML) remains a vital tool in assisting with malware detection, a major concern lies in its reliability and trustworthiness, particularly in safeguarding ML models against evasion attacks which entail crafting adversarial examples (AEs). Specifically, these attacks can deceive ML models by exploiting blind spots[1] in their decision space, where

This chapter is based on the paper currently under peer review: H. Bostani, J. Cortellazzi, D. Arp, F. Pierazzi, V. Moonsamy, and L. Cavallaro, Effectiveness of Adversarial Training on Malware Classifiers, 2025 [204].

[1] Blind spot is the informal name for AE [205].

the predictions are unreliable or inaccurate due to inadequate training samples [29]. While there exist several approaches for thwarting such attacks (e.g., defensive distillation [17], weight bounding [16, 28], and monotonic classification [49]), Adversarial Training (AT) [50]—also called adversarially robust optimization—remains one of the most successful defense strategies [51, 52]. Throughout the last decade, AT has been widely recognized as the primary solution to strengthen ML-based malware detection against evasion attacks [14, 17, 24, 25, 28–33, 35–38, 60, 137, 153, 154, 156, 206–208]; however, it has not been extensively investigated. Specifically, the efficacy of existing AT techniques in providing adversarial robustness against realistic evasion attacks while maintaining the original clean performance[2] remains an under-explored area.

In particular, we observed that several studies [14, 17, 24, 25, 29–33, 35–38, 60] employed improper evasion attacks to demonstrate the adversarial robustness derived from AT, because either the realizability of the AEs generated by their attacks is impractical [137] (as they do not meet all domain constraints formalized in [9]), or the evasion attacks considered for evaluation might not be strong enough to cause classification errors in the ML models [51]. Moreover, the robustness of AT explored in some studies, such as [206], was not evaluated in the worst-case scenario since they examined AT against only an evasion attack operated under Zero Knowledge (ZK) settings, which may be less effective than the ones operating under Perfect Knowledge (PK) settings. On the other hand, some studies [14, 17, 28, 30, 32, 37, 60, 206, 208] failed to consider effective hardening techniques for adversarial hardening, which might influence their conclusions, and consequently, limiting the generalizability of their findings.

Convergence [35] in solving the inner maximization problem is another aspect that the majority of studies [17, 24, 25, 28–33, 35, 37, 38, 60, 153, 156, 207, 208] mostly focus on while overlooking the exploration of other key dimensions (e.g., classifiers and feature representations) that affect the clean performance and adversarial robustness of ML models. Although exploring evasion attacks in AT can help mitigate convergence issues by producing high-confidence AEs (i.e., some AEs that are incorrectly classified with high confidence) [53], it remains uncertain whether these samples consistently improve the adversarial robustness of ML models utilized for malware detection. While [36] is the most concrete study that explored several settings in AT, it failed to comprehensively delve into the examination of attack perturbation bounds, AE confidence levels, and AE fractions that are deemed critical for maximizing the coverage of blind spots in AT [53, 209].

In this chapter, we propose a unified framework that enables a comprehensive investigation of how data, feature representations, classifiers, and robust optimization settings interact to shape the effectiveness of AT in malware detection. Using this framework, we uncover counterintuitive insights that challenge assumptions in prior work and offer several novel contributions, such as improving our understanding of how linear and non-linear models respond to AT. Specifically, building on prior research, our chapter revisits AT by questioning established assumptions and broadening exploration. Unlike earlier works that emphasize convergence issues [35], promote strict domain constraints [137, 206, 210], or narrowly target specific models like Bayesian Neural Networks [153], we adopt a more inclusive approach to challenge these ideas and explore new possibilities. Using our framework, we empirically show that AT's effectiveness in robust optimization depends on a complex

[2]The clean performance refers to vanilla malware classifiers' performance on clean data.

interaction of factors—such as perturbation bounds, adversarial confidence, adversarial fraction, and domain constraints—shaped by the underlying data, feature representations, and classifiers. Our results highlight the necessity of adopting a holistic methodology for AT, demonstrating that no single factor ensures robustness. Achieving optimal trade-offs between clean accuracy and robustness requires coordinated tuning of all parameters. Our **contributions** can be summarized as follows:

- We propose a unified framework (Section 6.3.3) that identifies key factors, such as the dimensionality of feature representations and AE realism, across multiple exploration and evaluation dimensions to examine how their interplay affects the effectiveness of AT. To support further research, we release our code at `https://github.com/HamidBostani2021/robust-optimization-malware-detection`.
- Our evaluation shows that clean and robust performance in hardened classifiers is shaped by the interplay of key factors. Challenging prior influential studies on AT [35, 36, 137, 153, 206, 210], we find that the success of AT is not absolute—it relies on careful tuning of parameters (e.g., type and quantity of AEs) alongside variations in data, feature representations, and classifiers.
- Through our empirical evaluations, we identified five evaluation pitfalls (Section 6.4.2.1) that impact current state-of-the-art research. Furthermore, we present ten key insights (Section 6.4.2.7) to guide researchers in refining their adversarial training methodology and deepening their understanding of the underlying principles.

6.2 Background

In this section, we briefly describe evasion attacks, and ML hardening.

6.2.1 Evasion Attacks

ML-based static analysis faces challenges such as evasion attacks, where the code is altered to evade detection without changing its functionality. The goal of an adversary in evasion attacks is to perform targeted or untargeted attacks to change the predicted class assigned by the classifier. In targeted attacks, the objective is to alter the prediction to a desired, predefined class; however, in untargeted attacks, any change in the predicted label from its original classification suffices. Evasion attacks can be categorized into feature-space attacks, which modify the input features, and problem-space attacks, which directly manipulate domain-specific objects [50, 164, 211], such as Android apps. The transformation between feature space and problem space is neither differentiable nor invertible, complicating the adversary's task in these dimensions.

Feature-space evasion attacks. These adversarial attacks operate directly on the feature vector representing the input data to the ML model. By making subtle modifications to the feature values, attackers can deceive the model into making incorrect classifications. This type of attack is particularly insidious because it requires knowledge of the model's features and how they are processed but does not necessarily require direct access to the model itself. The simplicity and effectiveness of feature-space attacks highlight the vulnerabilities inherent in relying solely on ML for security [212]. Feature-space evasion attacks can be categorized into constrained and unconstrained attacks. Constrained attacks target classification datasets

by considering inherent domain-specific constraints (e.g., feature immutability or non-linear relationships between features), while unconstrained feature-space attacks ignore these constraints.

Problem-space evasion attacks. These adversarial attacks involve manipulating the actual content of the input data, such as modifying an Android malware app without changing its malicious functionality. These attacks are more complex and require a deeper understanding of how modifications to the input data affect its representation in the feature space. Problem-space attacks are considered more practical from an attacker's perspective because they do not necessitate direct access to the model's internal workings. Instead, they focus on crafting inputs that are inherently challenging for the model to classify correctly [9].

From a general perspective, evasion attacks can be categorized as either realistic or unrealistic evasion attacks [137], regardless of where they are generated. Realistic evasion attacks generate realizable AEs by adhering to domain constraints in either the problem space or the feature space, where these representations resemble legitimate programs. Conversely, unrealistic evasion attacks produce AEs that may be unrealizable, meaning they do not resemble legitimate programs.

6.2.2 ML Hardening

To defend against evasion attacks, robust optimization techniques [53] have become a pivotal focus. These techniques are designed to enhance the resilience of ML models against adversarial attacks, which manipulate input data to cause misclassification. Among these techniques, adversarially robust optimization, also known as adversarial training, plays a crucial role in fortifying models by exposing them to AEs during the training phase. This exposure aims to improve the model's ability to generalize from adversarial perturbations it might encounter in real-world scenarios [53]. AT involves incorporating AEs into the training process, thereby enabling the model to learn from these perturbations and make more robust predictions. Specifically, the most established strategy [53] is to iteratively generate high-confidence AEs—worst-case AEs that cause the highest loss—and update the model parameters to minimize the classification error on these examples. This method has been shown to significantly enhance the model's resilience against certain types of attacks, though it may not guarantee protection against all possible adversarial inputs [50]. In addition to AT, adversarial retraining [71] follows a similar approach but is based on a simpler idea. This defense strategy involves enriching the training dataset from scratch with AEs generated by an evasion attack.

6.3 Methodology

This chapter highlights the intertwined factors crucial for effective AT. We first formulate the problem and define key dimensions, such as feature representations and classifiers that must be explored to understand AT's effectiveness. Then, we present a unified framework that introduces diverse training and evaluation factors to explore AT from various perspectives. Finally, we outline a systematic evaluation to understand how these factors impact adversarial robustness and malware classifier performance on clean data. Exploring AT through different training factors is vital, as practitioners must select appropriate configurations that will inherently influence their detectors' performance in real-world scenarios.

6.3.1 Problem Definition

Suppose Z represents the problem space encompassing all potential objects (e.g., Android apps or Windows programs). Additionally, F denotes the feature representation that characterizes the dimensions of the feature space X. For utilizing ML in malware detection, each object $z \in Z$ is first mapped to $x \in X$ through a mapping function $\psi : Z \to X$. Malware detection is then performed by a binary classifier $f : X \to Y$ with a discriminant function $g : X \times Y \to R$, where $f(x) = \arg\max_{i \in Y} g_i(x)$ determines the label of $x \in X$ from the label space $Y = \{0, 1\}$. Specifically, x is classified as malware if $f(x) = 1$, and as benign, otherwise.

Evasion attacks can transform a malware sample $x \in X$, which is correctly classified (i.e., $f(x) = 1$), into an AE by finding an adversarial perturbation δ that changes the prediction to $f(x + \delta) = 0$ when added to x. To identify δ, attackers solve the following optimization problem:

$$x' = \arg \max_{x' \in N(x)} \quad L(g_y(x'), \theta, y) \tag{6.1}$$

where $x' = x + \delta$, and L and θ denote the loss function and parameters of the classifier f, respectively. Moreover, $N(x)$ represents the set of allowed perturbations. The common constraint in finding δ, which represents the allowed perturbations, is a norm bound $\|x - x'\|_p \leq \epsilon$, where ϵ signifies the magnitude of the maximum allowed changes. To harden f against adversarial perturbations, robust optimization aims to adjust the parameters of f by incorporating eq. 6.1 in the training process and solving the following min-max optimization problem:

$$\min_{\theta} \mathbb{E}_{(x_i, y_i) \sim D} \left[\max_{\|x_i - x_i'\|_p \leq \epsilon} L(g_{y_i}(x_i'), \theta, y_i) \right] \tag{6.2}$$

where $\mathbb{E}$ denotes the expected value of the inner maximization problem, considering that $(x_i, y_i) \sim D$ are training data samples drawn from the distribution D. It is important to note that adversarial robustness implies that f can accurately classify any variations of x_i, demonstrated by $x_i' = x_i + \delta$, as long as $x_i' \in N(x_i)$. In other words, f exhibits adversarial robustness under $\|x_i - x_i'\|_p \leq \epsilon$ if $f(x_i) = f(x_i')$.

6.3.2 Characterizing the Effectiveness of AT

To assess the robustness of malware detectors derived from AT, it is crucial to initially identify the factors that might significantly impact the effectiveness of AT. Subsequently, comprehending how these factors influence both clean performance and the adversarial robustness of hardened malware detectors is essential. According to eq. 6.2, taking the following key dimensions into account is vital for identifying factors that are likely to significantly influence the performance of AT:

Data and Feature Representations. Eq. 6.2 shows that the training set is crucial for AT as robust optimization is performed in it. When the empirical distribution of the training set diverges from the true data distribution, AT may become ineffective since adversaries can generate AEs that fall outside the empirical distribution of the training set [192]. One of the primary dimensions significantly affecting the distribution of the training samples and the coverage of blind spots is feature representation. As shown in Figure 6.1, the distribution

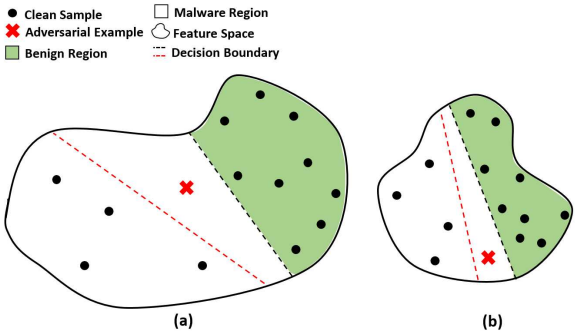


Figure 6.1: Illustrating the impact of feature representation on altering the data distribution and covering blind spots (i.e., the vulnerable regions between the decision boundary and the benign region).

of training samples and the size of the blind spots can be altered by using different feature representations. Specifically, in discrete feature spaces, using low-dimensional feature representations (e.g., Figure 6.1 (b)) can reduce the vulnerable region compared to high-dimensional feature representations (Figure 6.1 (a)), intuitively impacting the capabilities of uncovering blind spots by AT. Our experiments, especially the analysis in Section 6.4.2.2, tend to provide empirical insights into this matter.

Classifiers. Our work aims at understanding the role that different learning algorithms, especially linear and non-linear classifiers, play in model hardening through AT. The intuition is that low-flexible classifiers, such as linear SVM, might be more susceptible to adversarial instability compared to high-flexible classifiers [213], such as non-linear classifiers, resulting

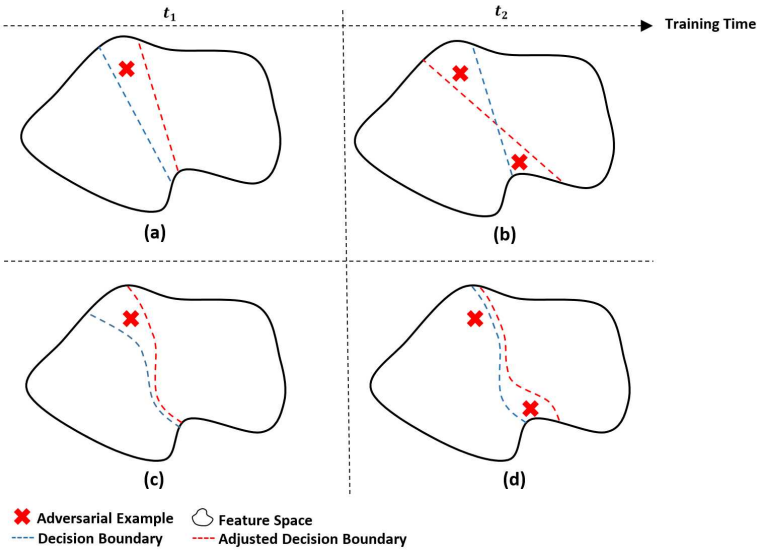


Figure 6.2: Demonstrating the impact of classifiers on AT: (a) and (b) show that a linear classifier, due to its low flexibility, may lose the adjustments made to its decision boundary at time t based on an AE when encountering a new AE at time $t + 1$. In contrast, (c) and (d) illustrate that a non-linear classifier is more adaptable when encountering new AE.

in varying levels of adversarial robustness. Specifically, as shown in Figure 6.2, linear classifiers might start to *forget* patterns of adversarial inputs encountered earlier in the training process due to their limited flexibility [154].

Robust Optimization Settings. Eq. 6.2 indicates that the following hyperparameters might have a considerable influence on the performance of AT:

(i) Perturbation bounds for identifying AEs and confidence levels of AEs. As shown in Figure 6.3 (a), intuitively a larger perturbation bound ϵ should uncover more blind spots when the adversarial example x' found in the inner maximization is misclassified with high confidence, achieving higher robustness.

(ii) Adversarial fractions, indicating the proportion of AEs used in AT. The robust optimization in eq. 6.2 utilizes malware samples, underscoring the critical importance of the number of malware samples used for AT. Increasing the number of AEs can potentially enhance AT's ability to uncover more blind spots. However, as illustrated in Figure 6.3 (b), this is more likely to occur if the original malware used for AT covers a broader feature space rather than just specific narrow areas.

(iii) Domain constraints. Realistic adversarial attacks target specific regions in the feature space to compromise ML-based malware detectors [137]. As shown in Figure 6.3 (c), this suggests that it is sufficient for AT to uncover only those vulnerable regions that include the feature representations of realizable AEs. These vulnerable regions can be exposed by considering domain constraints during AE generation, e.g., creating realizable AEs that satisfy the domain constraints specified in the problem space [9].

6.3.3 Unified Evaluation Framework

To thoroughly investigate AT, we propose an evaluation framework, shown in Figure 6.4, that helps identify impactful factors in AT. It allows for various controlled evaluations necessary to clarify the impact of the training factors within the key dimensions defined in Section 6.3.2

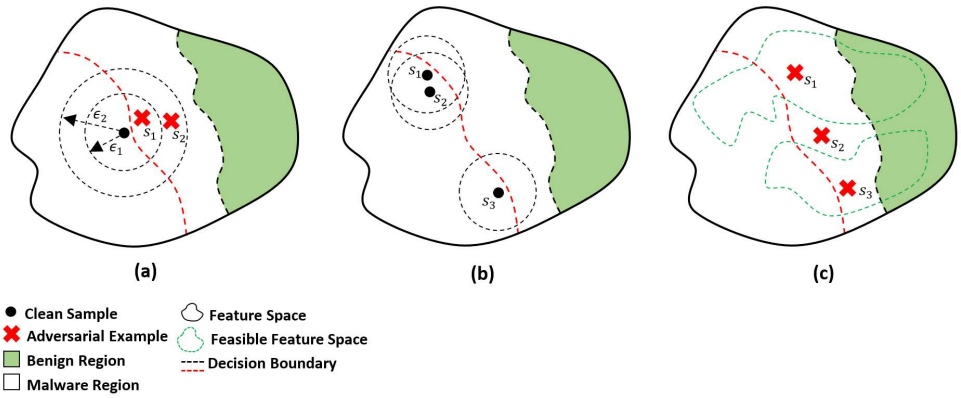


Figure 6.3: Illustrating the influence of different settings on the performance of robust optimization: (a) demonstrates that a large perturbation bound (e.g., ϵ_2), along with a high-confidence AE (e.g., s_2) can potentially reveal more blind spots. (b) shows that using different sets of malware samples results in varying effects on uncovering blind spots, e.g., s_1, s_2, and s_3 being more effective than $s1$ and s_2. (c) indicates that in AT, uncovering only those blind spots within the feasible feature space is sufficient, as realistic evasion attacks target these regions (e.g., s_1 and s_3 are realizable AEs, whereas s_2 is not realizable).

on the success of robust optimization. The framework defines these dimensions for AT and essential evaluations to assess vanilla and robust models, allowing systematic hypothesis testing and debugging of AT configurations through controlled factor adjustments.

6.3.3.1 Dimensions and Their Relevant Factors

The proposed framework facilitates investigating the effectiveness of AT based on various factors across the following defined key dimensions:

Data. *Distribution* and *volume* of data are shown to be two essential training factors in AT [74]. Our framework allows us to explore the impacts of the variations of these factors on the effectiveness of AT. Specifically, we can import various datasets of different sizes that include real-world objects like Android Packages (APKs), with diverse distributions based on variables such as source and release date. Additionally, we can specify the proportion of samples in the training, validation, and test sets.

Feature Representations. Our framework enables us to explore *dimensionality*, *sparsity*, and *types* of feature representations, as they seem to influence both model performance and computational efficiency. For instance, high-dimensional feature spaces can hinder generalization because increased features lead to sparser data points, which may cause models to capture incidental correlations instead of meaningful patterns [214]. It is important to note that by elucidating each supported feature representation within the framework, we ensure that all real-world objects in the training and test sets are represented in the feature space according to the specified representations.

Classifiers. The framework supports building various malware detectors by employing a range of learning algorithms. With support for both linear and non-linear classifiers, this key dimension facilitates a thorough investigation of how the *model flexibility*, which indicates the flexibility of classifiers, influences AT.

Robust Optimization Settings. *Adversarial confidence*, *perturbation bound*, *adversarial fraction*, and *domain constraints* are adjustable factors specific to AT. The framework helps us understand how variations in these factors, along with other discussed factors, contribute to strengthening malware classifiers. In the AT process, we can specify the perturbation bound for generating AEs and select different evasion attacks, either unrealistic feature-space attacks or realistic problem-space attacks, to solve the inner maximization problem in AT. The former is used to explore the influence of adversarial confidence, as different unrealistic feature-space attacks produce AEs with varying levels of misclassification confidence, while the latter is used to examine the impact of domain constraints since realistic problem-space attacks can generate realizable AEs that meet these constraints.

6.3.3.2 Evaluations

The proposed framework provides options for building either vanilla or robust malware detectors through standard or adversarial training. Exploring robust optimization settings is solely crucial for AT, while the remaining factors in the key dimensions are relevant for both types of training. Once ML models are built according to the configurations outlined in the framework's key dimensions, we can assess their performance using the following evaluation aspects. Please note that in Section 6.4.2.1, we discuss some pitfalls that might occur when exploring evaluation factors.

Clean Performance. Two important evaluation factors in specifying the performance of malware classifiers in the absence of adversarial attacks are *reliability* and *completeness*,

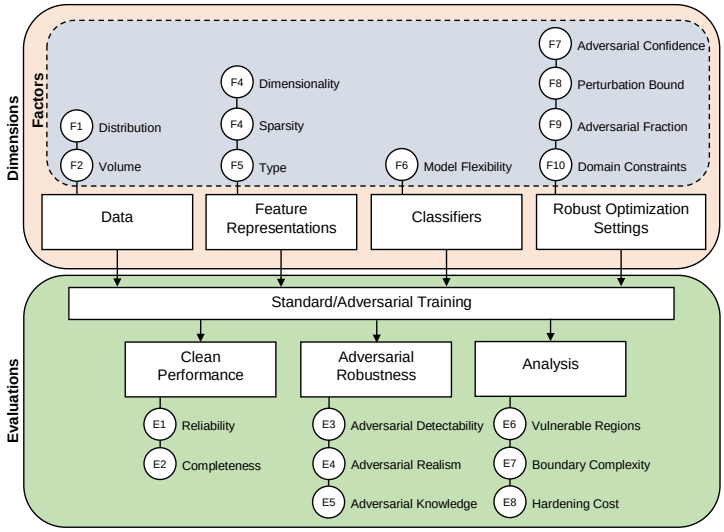


Figure 6.4: Illustration of our unified framework proposed to investigate the influence of various key dimensions on the performance of malware classifiers.

offering practical insight into their impact beyond formal definitions. Reliability ensures the classifier accurately detects malware without mislabeling goodware, while completeness ensures it identifies all malicious instances. The framework includes metrics like F1-Score to clarify these aspects.

Adversarial Robustness. Our proposed framework supports three key evaluation factors for assessing adversarial robustness: *adversarial detectability*, *adversarial realism*, and *adversarial knowledge*. While adversarial detectability measures the classifier's ability to detect AEs, adversarial realism ensures that only realistic evasion attacks are considered. Additionally, robustness is assessed against attacks with varying levels of knowledge about the classifier.

Analysis. The framework provides a collection of tools for plotting, such as t-SNE visualization [215]. Besides regular plots, two following tools are supported to further interpret the functionality of AT:

(i) Joint Feature Importance. This tool is designed to assess how *vulnerable regions* in the feature space are protected by AT. Specifically, to evaluate which features are important for both AT and a realistic evasion attack, we proposed a technique utilizing the Joint Distribution Plot (JDP). To this end, we first determine the frequency of alteration for each feature within the feature space during AT and attacking, as this frequency can be seen as a metric indicating the importance of features for either AT or the realistic evasion attack. For example, if $f_1 \in F$ is involved in transforming $x_1 \in X$ and $x_2 \in X$ into AEs during AT, its frequency is 2. We then use the JDP to visualize the overlap in features importance between AT and the evasion attack. The JDP is based on the Probability Distribution Function (PDF) and is plotted by estimating the values of two random variables: one representing the importance of features for AT and the other for the evasion attack. Using the PDF, the distribution of every feature is displayed along the x-axis and y-axis according to the frequencies determined by AT and

the evasion attack, respectively. Refer to *Section* 6.4.2.5 to see an example for JPD.
(ii) Decision-function Roughness. To investigate whether the adversarial vulnerability is related to *boundary complexity,* which indicates the shape of the decision surface learned by the models, we adapt the technique proposed in [209] for use in our framework. This method estimates a model's prediction-change risk r by comparing the predictions of synthetic samples, uniformly drawn within the ϵ-bound of a training sample $x \in X$, with the prediction of x. A larger r indicates a rougher decision function. For more details about this measurement, refer to [209].

Besides the above analysis factors, our framework includes the *hardening cost*, which captures the computational resources needed for AT, helping assess its practical feasibility.

6.3.4 Structured Analysis

To evaluate the influence of the key factors outlined in *Section* 6.3.2, we design a wide range of experiments that enable controlled assessments. Specifically, these experiments are crafted to explore the following RQs:

RQ1: What factors significantly affect the effectiveness of AT?

To explore this research question, we design a comprehensive set of experiments. Specifically, we systematically vary each of the different factors identified within our key dimensions to understand their contributions to the efficacy of AT. Our experimental design encompasses the following key aspects:

- Data: We consider different datasets including raw objects such as Android APKs, denoted as D_1, D_2, etc.
- Feature Representations: We use different feature representations, denoted as F_1, F_2, etc.
- Classifiers: We consider multiple classifiers, denoted as C_1, C_2, etc.
- Perturbation Bounds: We explore a range of perturbation bounds, such as $\mathcal{E} = [0, 5, 10, ..., 100]$.
- Adversarial Fractions: We vary the fraction of AEs, such as $A = [10\%, 20\%, ..., 100\%]$ where each $\alpha_i \in A$ indicates the percentage of malware samples per epoch that can be used for AT.
- Evasion Attacks: We employ several evasion attacks, donated as $Attack_1$, $Attack_2$, etc., to generate AEs required for either AT or attacking. Our attack set includes unrealistic feature-space attacks to explore the impact of AE confidence on AT, as well as realistic problem-space attacks to assess the effect of domain constraints on AT. It is noted that realistic problem-space attacks allow us to generate realizable AEs that satisfy domain constraints. Additionally, these problem-space attacks are used to evaluate the adversarial robustness of our classifiers.

For each combination of data, feature representation, and classifier, we conduct a series of experiments by varying the perturbation bounds, the adversarial fractions, and evasion

attacks. For instance, we first combine data D_1, feature representation F_1, and classifier C_1, and evaluate the performance at each $5 \in \mathcal{E}$ and $50\% \in A$ using different $Attack$. We then continue in this manner for all combinations.

RQ2: What properties of the generated AEs influence the outcomes of the hardening process?

Different evasion attacks generate AEs by leveraging distinct logical approaches, leading to varying levels of adversarial robustness. Therefore, each method uniquely influences the training process, resulting in different coverage of blind spots (i.e., vulnerable regions in the decision space of classifiers) and decision boundaries. To investigate this research question, we examine the coverage of blind spots and decision boundaries using analysis tools prepared in the framework, particularly Joint Feature Importance and Decision-function Roughness.

6.4 Experiments and Evaluations

6.4.1 Scope of Analysis

AT is platform-independent, operating on feature representations rather than raw problem-space data. However, to avoid biased datasets and enable thorough exploration, it is essential to collect a large, diverse set of malware and goodware samples. Android is well-suited for this, offering extensive timestamped APKs via repositories like AndroZoo [44], ensuring the volume and variety needed for reliable analysis.

We emphasize the necessity for practitioners to fine-tune key parameters (e.g., adversarial fraction) in robust optimization to ensure its effectiveness. Our framework underscores the importance of understanding how these parameters interact with variations in data, feature representations, and classifiers. Given the challenges in identifying optimal feature representations and classifiers, we adopt well-established solutions to maintain focus on our primary research objectives. Specifically, to explore training factors such as *feature dimensionality* and *model flexibility*, we utilize multiple datasets (DREBIN20 [9] and APIGraph [54]), feature representations (DREBIN [96] and RAMDA [159]), classifiers (linear Support Vector Machine (SVM), Decision Tree (DT), and Deep Neural Network (DNN)). To address evaluation factors like *adversarial realism* and *adversarial knowledge*, we consider both unrealistic (PGD [53], JSMA [164]) and realistic (PK-Greedy [9], EvadeDroid [84]) attacks, generating AEs from 1K randomly selected clean malware samples from the test set (true positives). Details of our choices and implementation are provided in Appendices 6.A and 6.B.

6.4.2 Systematic Evaluations

This section explores RQ1 in Section 6.4.2.2, Section 6.4.2.3, and Section 6.4.2.4, as well as RQ2 in Section 6.4.2.5 to determine the impact of various key factors on the success of AT and explore which properties of AEs influence AT performance. As outlined in Section 6.3.4, our systematic evaluations are structured around a series of experiments, each employing various combinations of data, feature representations, and classifiers.

6.4.2.1 Standard Evaluation Configuration

PGD and JSMA are two feature-space attacks within the optimization module of our proposed framework. They are used to harden baseline malware detectors in most of our evaluations by generating adversarial examples (AEs) with varying confidence levels (see Appendix 6.C). Additionally, given that both DREBIN and RAMDA are binary feature representations, we consider the ℓ_0-norm (i.e., the number of changes) in eq. 6.2, because it is the common perturbation bound for binary feature representations [108]. Moreover, to assess adversarial robustness, we evaluate the models' resistance to bounded PK-Greedy and EvadeDroid attacks, ensuring that their attack bounds match those used in the evasion attacks during AT. Additionally, it is imperative to confirm that the observed adversarial robustness against evasion attacks is attributable to AT, rather than stemming from the limitation imposed by the attack bounds of evasion attacks. Our preliminary evaluations in Appendix 6.D underscore the importance of excluding the robustness improvement resulting from the limitations imposed by the attack bound when assessing the adversarial robustness of a model enhanced with AT. This approach is crucial for accurately understanding the impact of AT on the model's robustness. In particular, consider M_v and M_h, representing a vanilla model and its hardened counterpart strengthened with an evasion attack employing an ϵ bound during AT. To assess the robust accuracy solely obtained through AT of M_h against an ϵ-bounded evasion attack $Attack$, the measurement should be conducted as follows:

$$R_{\text{AT}} = R_h - (R_{v_1} - R_{v_2}) \tag{6.3}$$

where R_h represents the robust accuracy of M_h in response to the ϵ-bounded $Attack$. Additionally, R_{v_1} denotes the robust accuracy of M_v under the same ϵ-bounded attack, while R_{v_2} indicates M_v's robust accuracy against the unbounded $Attack$. While R_{AT} isolates the robustness directly attributed to AT, comparing its effectiveness across different *attack bounds* requires normalization, as AT does not have the same opportunity to improve robustness when the inherent robustness of the vanilla model varies. For instance, with a smaller attack bound, the vanilla model may already exhibit higher robustness, limiting the potential gain from AT. We define the relative robustness gained by AT as:

$$R_{\text{rel}} = \frac{R_{\text{AT}}}{100 - (R_{v_1} - R_{v_2})} \times 100 \tag{6.4}$$

where $100 - (Rv_1 - Rv_2)$ represents the portion of robustness that is still available for improvement by AT in M_h under the ϵ-bounded $Attack$. Here, 100 represents the theoretical maximum robustness a model can achieve. This normalization ensures a fair comparison across different attack bounds by measuring how effectively AT improves robustness relative to the remaining improvable portion. The aforementioned evaluations give rise to Pitfalls 1 and 2, which are described as follows:

Pitfall 1—Overestimated Robustness. It is common to measure the adversarial robustness of malware detectors after hardening ML models with AT. However, overlooking the robustness arising from the limitations of adversaries due to the attack bound could result in an overestimation when reporting the adversarial robustness achieved through AT.

Recommendation for Pitfall 1. To verify the efficacy of AT methods in enhancing adversarial robustness, eq. 6.3 can be employed to discount the initial robustness of baseline malware detectors against bounded evasion attacks. Figure 6.5 provides an example highlighting

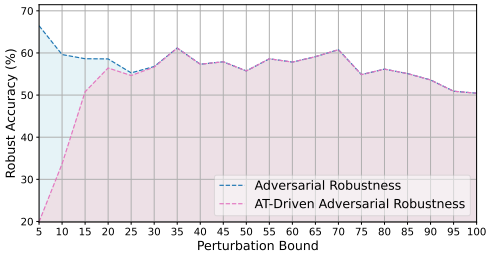


Figure 6.5: An example demonstrating the robust accuracy of hardened DNN models trained on RAMDA with varying perturbation bounds, evaluated against PK-Greedy attacks with identical attack bounds. The models are strengthened by utilizing PGD in AT.

the difference between the measured adversarial robustness and the adversarial robustness derived from AT (as identified by eq. 6.3) for various DNN models trained on RAMDA.

Pitfall 2—Limited Threat Models. Investigating AT could result in misleading conclusions if the robust evaluation of malware detectors is performed using only one threat model. For example in [206], malware classifiers have been compared against a single attack, which may lead to an unreliable conclusion. Our observation in Table 6.4 of Appendix 6.D confirms that malware detectors provide various robustness against attacks with different threat models. For instance, Table 6.4 intriguingly indicates that EvadeDroid, which operates in ZK settings, outperforms PK-Greedy, which operates in PK settings.

Recommendation for Pitfall 2. The evaluation of the adversarial robustness of detectors against evasion attacks should involve considering different threat models, including at least PK and ZK evasion attacks. This is because of robustness against PK attacks do not necessarily imply robustness against ZK attacks, and vice versa, as they might target different vulnerable regions.

Additionally, we identified the following three pitfalls during our preliminary assessment aimed at ensuring the framework's effectiveness.

Pitfall 3—Reproducibility Challenges. ML algorithms often involve random processes, such as weight initialization and data shuffling. For instance, our preliminary evaluation illustrates the F1 score of the same model adversarially trained with the same varied perturbation bounds in two runs is different. We cannot ensure that the changes in the F1 score are due to the perturbation bound alone, as the scores differ even for a single perturbation bound.

Recommendation for Pitfall 3. Since results can vary between runs without consistently setting random seeds, it is essential to fix the random seed for every stochastic operation in the training process. This ensures that, at least on the same machine, observed differences are due to variations in AT settings rather than randomness.

Pitfall 4—Role of Representations. Multiple representations may be available for the data considered. For instance, for malware, we have multiple available feature spaces in the research community. Nevertheless, research frequently focuses on a single feature space, which could have inherent limitations and may not be the best choice for the considered task.

Recommendation for Pitfall 4. Since intrinsic characteristics of different representations may vary, it would be worth exploring multiple representations and then proceeding with the most suitable for the considered task.

Pitfall 5—Adversarial Realism Challenge. In the real world, not all adversarial attacks are

feasible, as adversarial malware aims to both bypass malware classifiers and compromise victim machines. Therefore, using evasion attacks that cannot generate realizable AEs for robustness evaluation may lead to misleading results.

Recommendation for Pitfall 5. When evaluating adversarial robustness, it is necessary to consider evasion attacks that are realistic by satisfying domain constraints, either in the problem space [9] or in the feature space [137].

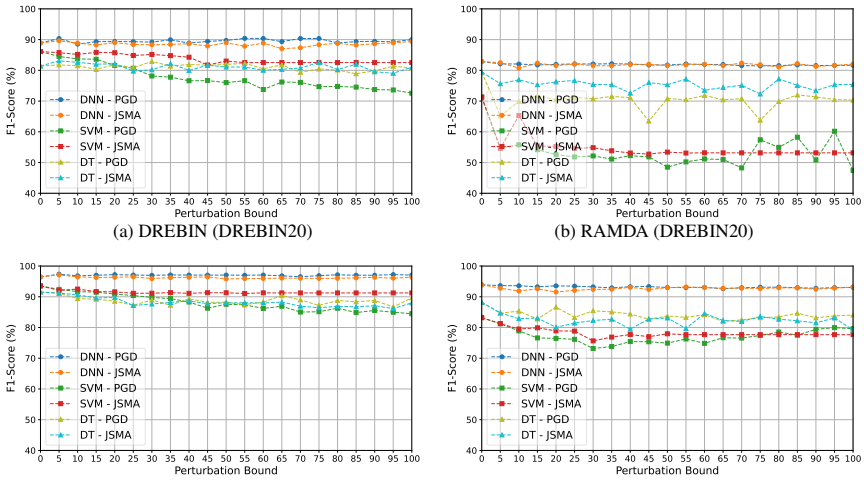
6.4.2.2 Robust Optimization Settings: Variation of Perturbation Bounds and AE Confidence Levels

To investigate the impact of these two variables on AT, we utilized PGD and JSMA to generate AEs with perturbation bounds varying from 5 to 100 in increments of 5. Additionally, we set the fraction of AEs to $\alpha = 50\%$. Figure 6.6 shows the clean performance of different models in terms of the F1 score. As can be seen, different classifiers exhibit varying sensitivities to AT with changing perturbation bound ϵ and the confidence of AEs. Increasing ϵ potentially enables the inner optimization to find AEs with higher confidence, provided the evasion attack used in AT is able to generate high-confidence AEs. As shown in Figure 6.6, using AT for hardening linear SVM and DT often compromises clean performance, with a greater sacrifice as ϵ increases, especially when PGD is used for hardening linear SVM. In contrast, DNN is more adaptable. For instance, Figure 6.6 (a) depicts in DREBIN (DREBIN20), the F1 Score of linear SVM significantly decreases from 86.1% to 72.6% when PGD is used in AT with $\epsilon = 100$. Using JSMA in AT has a lower effect on clean performance than PGD, especially in linear SVM, as the attack might not generate high-confidence AEs even with increasing ϵ. For instance, as can be seen in Figure 6.6 (b), while the F1 Score of linear SVM trained on RAMDA drops significantly with JSMA at $\epsilon = 5$, it remains relatively stable for higher $\epsilon = 5$ because, as shown in Figure 6.11 of Appendix 6.C, the confidence of AEs generated by JSMA does not significantly change with increasing perturbation bound.

Figure 6.7 illustrates the relative robust accuracy of different models trained on DREBIN and RAMDA against PK-Greedy and EvadeDroid on DREBIN20 and APIGraph datasets. Although Figure 6.7 shows variations in the adversarial robustness of models when hardened with different perturbation bounds, our observations reveal a few interesting results. First, models trained on RAMDA (dense, low-dimensional discrete feature space) often exhibit higher adversarial robustness compared to those trained on DREBIN (sparse, high-dimensional discrete feature space), as AT can potentially uncover more vulnerable regions in the lower-dimensional space. For example, Figure 6.7 (a) shows that 12 out of 20 DNN-JSMA models trained on DREBIN achieve a robust accuracy greater than 50% against PK-Greedy, whereas as can be seen in Figure 6.7 (c), 18 models achieve this benchmark when trained on RAMDA.

Second, low-confidence AEs like those from JSMA often enhance the robustness of linear SVM on RAMDA as the perturbation bound increases (Figure 6.7 (c, d, g, and h)). This is likely because, in dense, low-dimensional spaces, AEs within smaller ϵ-bounded regions have limited impact on the decision boundary. Larger ϵ enables greater shifts, but at the cost of clean performance. A similar pattern is observed for linear SVM on DREBIN, a sparse, high-dimensional space, when high-confidence AEs like those from PGD are used in AT (Figure 6.7 (b, e, and f)).

Thirdly, incorporating high-confidence AEs in AT does not consistently lead to improved adversarial robustness. For instance, as depicted in Figure 6.7 (a to d), among 240 models



(a) DREBIN (DREBIN20) (b) RAMDA (DREBIN20)

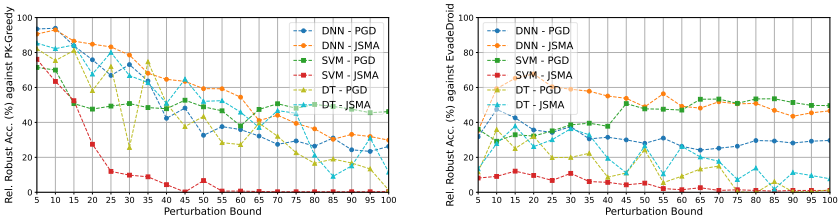
(c) DREBIN (APIGraph) (d) RAMDA (APIGraph)

Figure 6.6: Clean performance of various models trained on (a) the DREBIN and (b) RAMDA representations of the DREBIN20 dataset, and (c) the DREBIN and (d) RAMDA representations of the APIGraph dataset, measured in terms of F1 score. The models are strengthened using either PGD or JSMA with different perturbation bounds. The F-scores of different vanilla models are displayed with a perturbation bound of 0.

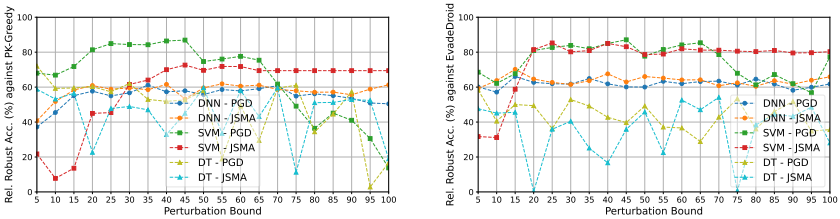
strengthened with JSMA, 143 models demonstrate higher adversarial robustness against realistic evasion attacks compared to those strengthened with PGD. As another example, Figure 6.7 (e to h) shows that using JSMA (low-confidence AEs) for hardening DNNs on both DREBIN and RAMDA representations of the APIGraph dataset often provides better adversarial robustness. However, within linear SVM, employing high-confidence AEs generated by PGD during AT, particularly in models trained on DREBIN, enhances adversarial robustness more effectively than JSMA. This observation appears to correlate with the linear decision boundary learned during training, which can be significantly shifted by high-confidence AEs—those characterized by high loss—thereby exposing larger vulnerable regions. Nevertheless, as illustrated in Figure 6.6, these substantial adjustments come at the cost of notable reductions in clean performance. It is noted that Appendix 6.E examines larger perturbation bounds for DREBIN, as the current bounds may be insufficient for this high-dimensional space. We observe that using attacks with large perturbation bounds in AT yields minimal to no effect, or even a negative impact, across all models.

6.4.2.3 Robust Optimization Settings: Variation of AE Fractions and AE Confidence Levels

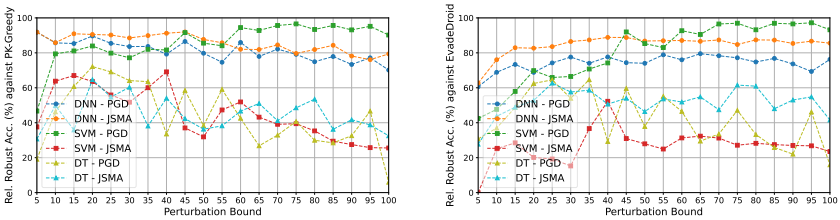
Increasing the number of AEs can potentially uncover more blind spots if the malware samples in the training set are uniformly distributed around the decision boundary; however, they might hurt the clean performance. In this experiment, we explore the impact of different fractions of AEs utilized in AT on both clean and robust accuracy. Specifically, the experimental design is similar to Section 6.4.2.2, but in this case, we set $\epsilon = 50$ and vary the AE fraction from 10% to 100% in increments of 10%. The relative robust accuracy is shown in Figure 6.15 of Appendix 6.G and the clean accuracy is reported in Figure 6.14 of Appendix 6.F. As with the previous evaluations, the plots demonstrate the significance of



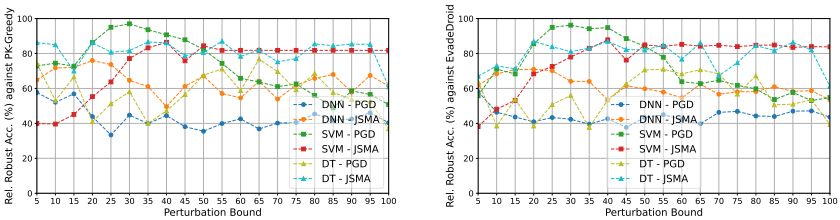
(a) DREBIN (DREBIN20) – PK-Greedy (b) DREBIN (DREBIN20) – EvadeDroid



(c) RAMDA (DREBIN20) – PK-Greedy (d) RAMDA (DREBIN20) – EvadeDroid



(e) DREBIN (APIGraph) – PK-Greedy (f) DREBIN (APIGraph) – EvadeDroid



(g) RAMDA (APIGraph) – PK-Greedy (h) RAMDA (APIGraph) – EvadeDroid

Figure 6.7: Relative clean accuracy improvement from AT on hardened models. Subfigures (a) and (b) show results on the DREBIN representation of the DREBIN20 dataset; (c) and (d) show RAMDA representation of DREBIN20; (e) and (f) show DREBIN representation of the APIGraph dataset; and (g) and (h) show RAMDA representation of APIGraph. Each model is hardened using either PGD or JSMA with varying perturbation budgets and evaluated against PK-Greedy and EvadeDroid attacks.

classifiers in AT. Non-linear models, such as DNNs, demonstrate adaptation of their decision boundaries to AEs, with minimal impact on clean accuracy across both feature spaces. In contrast, linear SVM exhibits a notable enhancement in robust accuracy, exceeding 60% in the DREBIN feature space. However, this improvement comes at the cost of a significant decrease in clean performance as the number of AEs during training increases. It is worth noting that this exploration indicates that the issue of low adversarial robustness with large

perturbation bounds in non-linear models, highlighted in Section 6.4.2.2, can be mitigated by increasing the number of AEs used in AT as the robustness of DNN models often improves with a higher adversarial fraction.

6.4.2.4 Robust Optimization Settings: Domain Constraints

The domain constraints can guide AT to focus on regions vulnerable to realistic evasion attacks. To examine the impact of these constraints on AT's success, we used realizable AEs generated by bounded PK-Greedy and bounded EvadeDroid in our AT process. Since employing problem-space adversarial attacks for robust optimization to solve the inner maximization problem significantly increases training time, we only consider the DREBIN20 dataset in our evaluations. Table 6.1 shows the settings for the robust optimization considered in this experiment. Our criterion for selecting the perturbation bound ϵ is to ensure that PK-Greedy and EvadeDroid achieve maximum success rates in fooling the evaluated models.

Figure 6.8 illustrates the robust accuracy of various malware classifiers hardened with unrealistic and realistic evasion attacks. Our observations for DREBIN indicate that realistic evasion attacks often provide robustness against similar attacks. For instance, as shown in Figure 6.8 (DREBIN-DNN, subfigure a), while utilizing PK-Greedy achieves relatively high adversarial robustness against PK-Greedy (63.6% robust accuracy), its robustness against EvadeDroid is very low (e.g., 0.5% robust accuracy). However, employing unrealistic evasion attacks such as JSMA can strengthen DNN models against both realistic evasion attacks, achieving 59.3% robust accuracy against PK-Greedy and 48.6% against EvadeDroid. We observe a similar trend for other classifiers, but the robustness achieved using PK-Greedy and EvadeDroid in AT is significantly lower for SVM and DT compared to DNN. Overall, PK-Greedy and EvadeDroid are less effective at hardening malware classifiers trained on DREBIN (a sparse, high-dimensional discrete feature space), particularly for models using SVM and DT. However, these results highlight that domain constraints, which are implicitly defined through the use of realizable AEs in AT, have a more significant impact on the success of AT in dense, low-dimensional discrete feature spaces. For instance, applying EvadeDroid to harden an SVM trained on RAMDA resulted in 41.8% robustness against PK-Greedy, while the same model trained on DREBIN only achieved 0.5%.

Furthermore, Figures 6.8 (RAMDA-DNN, subfigure a and d) and 6.8 (RAMDA-DT, subfigure a and d) show that utilizing EvadeDroid to harden DNN and DT models trained on RAMDA provides high adversarial robustness against both PK-Greedy and EvadeDroid. This suggests that exploring the low-dimensional feature space blindly is more effective than following gradients, which are potentially biased towards finding certain vulnerable regions rather than all vulnerable regions. In other words, in the RAMDA, which is a

Table 6.1: Parameter settings for domain constraints exploration in terms of adversarial rate α and perturbation bound ϵ. PG and ED define PK-Greedy and EvadeDroid, respectively.

Model	α	ϵ (DREBIN)		ϵ (RAMDA)	
		PG	ED	PG	ED
DNN	50	50	80	90	50
SVM	50	100	65	25	65
DT	50	70	85	75	30

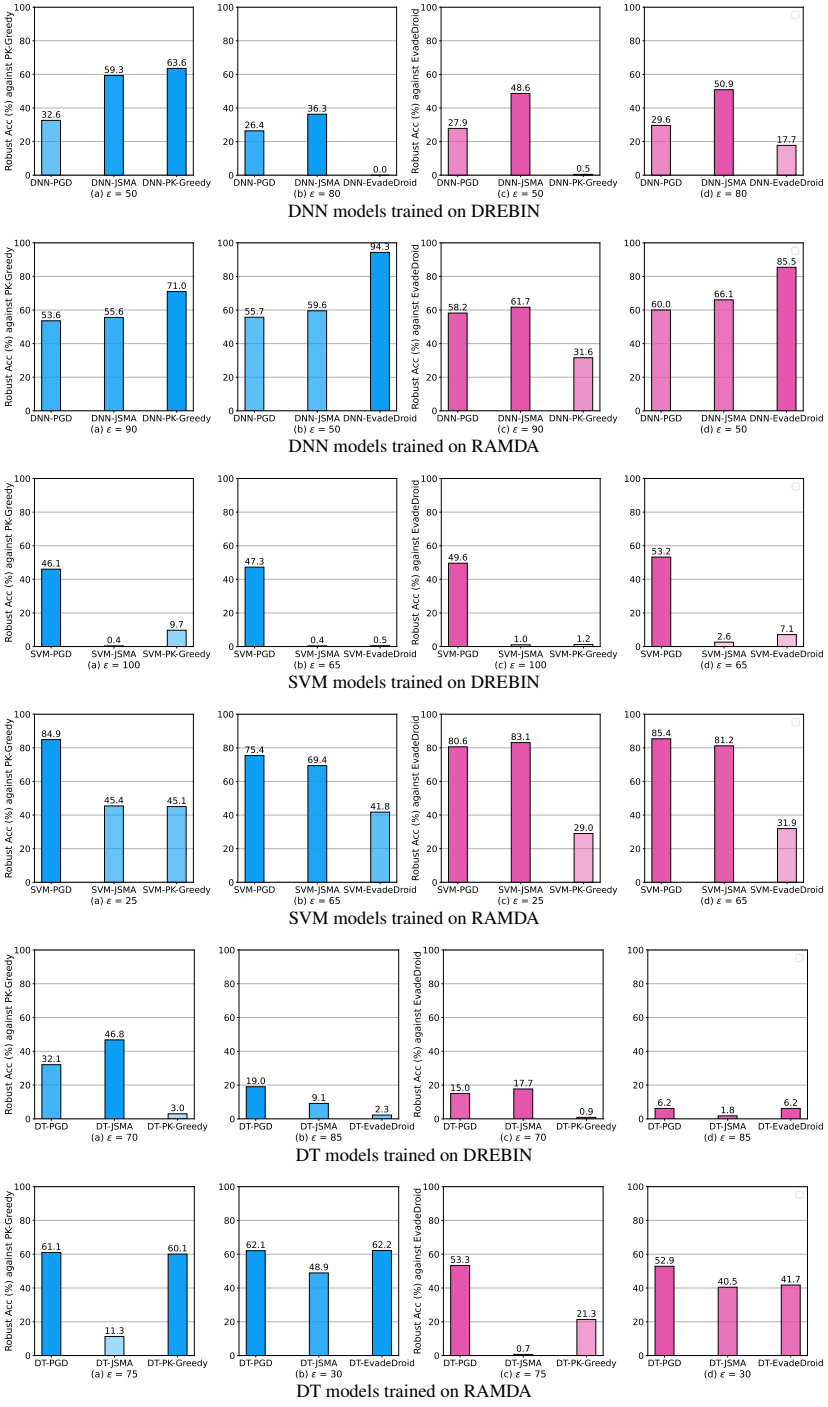


Figure 6.8: Robust accuracy, gained from AT, of DNN, SVM, and DT models trained on DREBIN and RAMDA representations of DREBIN20, hardened with either unrealistic or realistic evasion attacks against realistic evasion attacks. The perturbation bound and attack bound, both represented by ϵ, are identical in each subfigure.

Table 6.2: Clean performance of different models trained on DREBIN and RAMDA representations of DREBIN20, hardened with either unrealistic or realistic evasion attacks in terms of F1 Score. PG and ED define PK-Greedy and EvadeDroid, respectively. F1 scores of vanilla DNN, SVM, and DT trained on DREBIN representation are 88.9, 86.1, and 81.4, respectively. F1 scores of vanilla DNN, SVM, and DT trained on RAMDA representation are 82.8, 71.3, and 79.5, respectively.

Model	DREBIN				RAMDA			
	ϵ	PGD	JSMA	PG	ϵ	PGD	JSMA	PG
DNN	50	89.7%	89.0%	89.1%	90	81.5%	81.3%	82.6%
SVM	100	72.6%	82.5%	85.1%	25	51.8%	54.6%	70.9%
DT	70	79.4%	80.8%	82.1%	75	63.8%	72.3%	77.5%
	ϵ	PGD	JSMA	ED	ϵ	PGD	JSMA	ED
DNN	80	88.9%	88.8%	90.4%	50	81.7%	81.5%	82.3%
SVM	65	76.2%	82.5%	85.5%	65	51.0%	53.1%	71.2%
DT	85	78.9%	82.0%	83.6%	30	70.7%	75.4%	78.6%

low-dimensional feature space, robustness achieved by EvadeDroid, which works in ZK settings, is more transferable than PK-Greedy which works in PK settings.

Our observations in Figures 6.8 indicate that PGD and JSMA can potentially be effective in hardening malware classifiers; however, their effectiveness must be weighed against their impacts on clean performance. Table 6.2 shows notable drops in F1 Score, especially when PGD and JSMA are used for hardening SVM and DT, while PK-Greedy and EvadeDroid maintain clean accuracy. The decline in clean performance with unrealistic evasion attacks stems from AEs distorting class boundaries, causing artificial overlaps. This shift forces the model to focus on unfeasible regions (i.e., the areas where realizable AEs cannot be placed), hindering its ability to generalize and leading to more misclassifications on clean data. The use of realizable AEs in AT avoids these negative effects by guiding the optimizer to explore feasible and genuinely vulnerable regions.

6.4.2.5 Properties of AEs

This section investigates RQ2. This research question is similar to the research question explored in [206]. Intuitively, different methodologies for crafting AEs inherently produce distinct characteristics in the generated samples, as they exploit varying sets of features and algorithms. This distinction is illustrated in Figure 6.11 of Appendix 6.C, where it is evident that the confidence levels of AEs created by two different gradient-based strategies vary significantly. To delve deeper into AEs and their influence on the effectiveness of AT, we employ diverse tools mentioned in Section 6.3.3.2.

We first utilize the joint feature importance metric introduced in 6.3.3.2 to determine the significance of features for both the defense and attack sides. Specifically, we analyze some interesting results reported in Section 6.4.2.4. As shown in Figures 6.8 (b) and 6.8 (d) for DNN models trained on RAMDA, the adversarial robustness achieved by using EvadeDroid in hardening DNN trained on RAMDA is highly transferable, providing high adversarial robustness against not only EvadeDroid but also PK-Greedy. Figures 6.9 (a) and 6.9 (b) clarify this, demonstrating that some overlapping regions are highly important for both the defender and the attacker. Note that in each plot, the contours highlight regions where the

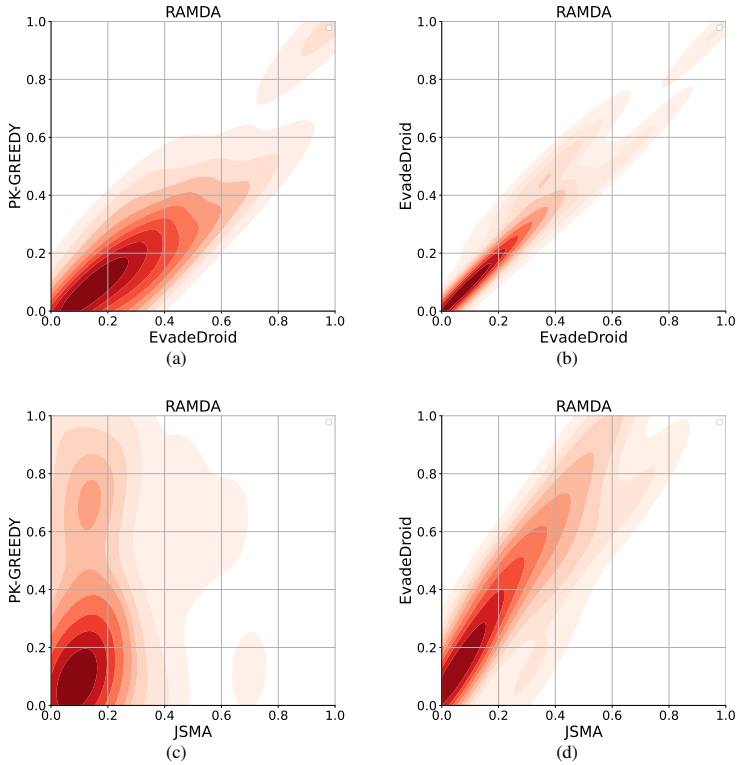


Figure 6.9: Joint distribution plots which compare the importance of features used both during the hardening phase (x-axis) and the attack phase (y-axis), referring to the scenario described in Figure 6.8.

importance of common features is greater than in areas outside the contours. The contours near the top right indicate that the common features are mostly important for both the defender and the attacker, with darker contours representing a higher feature overlap. Indeed Figures 6.9 (a) and 6.9 (b) demonstrate that EvadeDroid can generate some AEs during training that highlight features often targeted by both EvadeDroid and PK-Greedy during attacks. This helps the DNN model guard against adversarial changes that might affect these features. Figures 6.8 (b) and 6.8 (d) for DNN models trained on RAMDA also illustrate a relatively high adversarial robustness achieved by hardening DNN with JSMA, an unrealistic evasion attack. We visualize similar joint importance feature plots (Figures 6.9 (c) and 6.9 (d)) which, like the previous analysis, demonstrate that some overlapping regions are highly important for both defense and attack sides.

Then, we examine the potential connection between the roughness of the decision boundary, which is measured using the decision-function roughness metric introduced in Section 6.3.3.2, and adversarial robustness. Here, we consider several models hardened by PGD on RAMDA with ϵ values ranging from 50 to 60. As shown in Figure 6.7 (c), this range is particularly interesting because, at $\epsilon = 55$, DT exhibits a significant drop in adversarial robustness against PK-Greedy, while DNN and SVM models remain more stable.

Table 6.3: Analyzing various models trained on RAMDA in terms of decision-function roughness γ and robust accuracy R_Acc against PK-Greedy. These models are hardened by PGD.

Model	$\epsilon = 50$		$\epsilon = 55$		$\epsilon = 60$	
	γ	**R_Acc**	γ	**R_Acc**	γ	**R_Acc**
DNN	0.82	55.7%	0.82	58.6%	0.84	57.8%
SVM	0.26	74.6%	0.31	76.0%	0.26	77.6%
DT	0.16	58.1%	0.13	18.8%	0.17	49.2%

Table 6.3 presents the decision-function roughness γ and robust accuracy for these models. Our observations indicate that increasing ϵ can affect adversarial robustness, particularly in DT. For instance, DT's robust accuracy drops significantly from 58.1% to 18.8% when ϵ changes from 50 to 55, whereas the γ changes only slightly by -0.03. Furthermore, as shown for the DT model, increasing or decreasing ϵ does not necessarily correlate with changes in γ. Lastly, while the results for DNN and SVM suggest that lower γ can lead to better adversarial robustness, this is not consistently the case, as evidenced by the DT results contradicting this hypothesis.

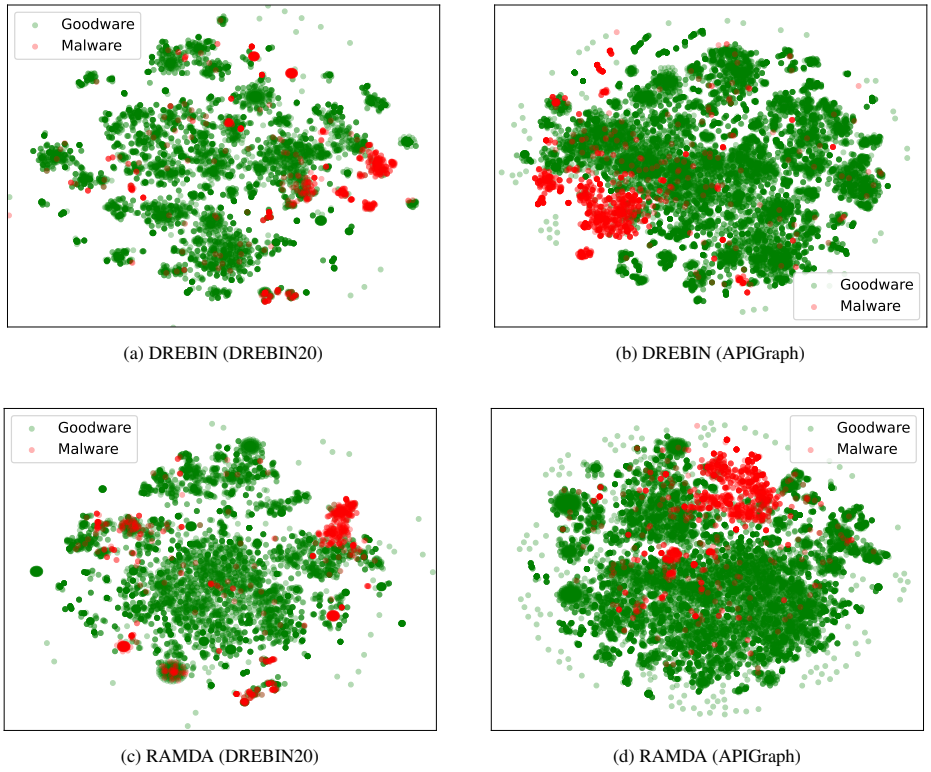
Next, as shown in the results presented in Section 6.4.2.2 and Section 6.4.2.3, hardened models trained on the APIGraph dataset often achieve higher clean and robust accuracy compared to those trained on DREBIN20. To understand this performance gap, we use t-SNE visualizations of both datasets. As shown in Figure 6.10, APIGraph plots—under both DREBIN and RAMDA features—demonstrate clearer separation between malware and goodware than DREBIN20, likely contributing to the improved performance. Additionally, prior work [216] indicates that APIGraph experiences less distribution drift, supporting its suitability for training effective and stable malware classifiers.

6.4.2.6 Computational Constraints

This chapter systematically analyzes how various factors affect the effectiveness of AT in malware classification. We trained 1K+ models and ran 3K+ evaluations, which took more than six months despite significant hardware resources (see Appendix 6.A.5). Specifically, training times ranged from 4 to 140 hours per model, depending on the settings, while clean performance and robustness evaluations took 15 and 780 seconds per model, respectively. It is important to note that a full exploration of all possible combinations—datasets, feature representations, classifiers, and robust optimization settings (e.g., perturbation bounds and adversarial fractions)—would require nearly 27K models (see Appendix 6.H). Given that training just 1K models took over six months, pursuing an exhaustive training regimen is computationally infeasible. To balance practicality and meaningful insights, we adopt a structured approach, varying key parameters while holding others constant. For instance, in Section 6.4.2.2, we examine perturbation bounds and adversarial confidence across datasets, features, and classifiers, fixing the adversarial fraction and using unrealistic evasion attacks, reducing the model count to 480 for efficient evaluation.

6.4.2.7 Key Findings

Our results shed light on the trade-offs between clean performance and adversarial robustness when using AT to strengthen malware classifiers, advocating for structured evaluations



(a) DREBIN (DREBIN20)

(b) DREBIN (APIGraph)

(c) RAMDA (DREBIN20)

(d) RAMDA (APIGraph)

Figure 6.10: t-SNE plots of goodware and malware from DREBIN20 and APIGraph, represented with DREBIN and RAMDA features.

over exhaustive training. Our chapter challenges conclusions drawn in previous influential research: Specifically, Section 6.4.2.4 questions the conclusions of [206, 210], arguing that realistic evasion attacks do not always enhance robustness, as their effectiveness depends on multiple factors. Additionally, while [36, 210] report that unrealistic evasion attacks degrade clean performance, we show that this effect varies by classifier, particularly in deep, non-linear models, where AT with unrealistic evasion attacks does not necessarily harm clean performance. However, when realistic evasion attacks are used, our results align with [36, 210], supporting the claim that such settings can maintain clean accuracy. Our findings in Section 6.4.2.2 and Section 6.4.2.3 also challenge the assumption made in [35], demonstrating that generating highly confident AEs in the inner loop of adversarially robust optimization is not always necessary. In fact, depending on the specific settings, using low-confidence AEs—settling for a local rather than a global solution—can sometimes lead to better robustness. Finally, our results in Section 6.4.2.3 challenge the findings of [206] regarding the adversarial fraction, showing that AT can benefit from increasing the budget of AEs generated by unrealistic evasion attacks.

In addition to challenging the relevant important studies, our systematic evaluations reveal new insights in malware detection. Drawing on our extensive empirical investigation, whose complete results are summarized in Table 6.5 of Appendix 6.I, we identify ten key

findings to inform and guide future research on AT in the malware domain.

> **Takeaway 1:** Linear and shallow non-linear models hardened with AT on discrete feature space substantially lose their performance on clean data, whereas deep non-linear models do not.

Our observations in Section 6.4.2.2 as well as Section 6.4.2.3 demonstrate that the clean performance of SVM (linear model) and DT (shallow non-linear model) hardened by AT is often notably sensitive to changes in the perturbation bound. Specifically, larger perturbation bounds result in greater sacrifices in clean performance, and vice versa. The sensitivity, especially in linear models decreases when AEs with lower confidence are used in AT. Conversely, adjustments to the perturbation bound have minimal impact on the clean performance of deep non-linear models.

> **Takeaway 2:** The amount and distribution of adversarial examples matters.

As demonstrated in Section 6.4.2.3, the fraction of AEs utilized in AT, specifically in each batch of robust optimization is crucial as it impacts the amount of considered AEs and the covered distribution. A minimum threshold is required to effectively shape a robust decision boundary, but it is important not to exceed this threshold in order to not heavily impact the clean accuracy. Conversely, using the entire batch of adversarial examples can degrade classifier performance, particularly in linear models. It is important to note that this finding validates similar results discussed in [209], which were explored in other domains.

> **Takeaway 3:** In discrete feature spaces, low-dimensional dense feature representations are easier to harden than high-dimensional sparse feature representations.

As illustrated in Figures 6.7 and 6.15, in discrete feature space, low-dimensional dense feature representations demonstrate superior robust accuracy compared to high-dimensional sparse feature representations across the tested classifiers, thereby revealing more vulnerabilities during AT. This can also be highlighted based on the transferability property of hardened models with problem-space AEs. Figure 6.8 further confirms that in low-dimensional feature spaces, realistic evasion attacks—constrained adversarial attacks—can effectively explore regions akin to other attack types. However, achieving such transferability is less pronounced and more challenging in high-dimensional, sparse feature spaces.

> **Takeaway 4:** Moving toward unbounded perturbations does not necessarily lead to any benefit in AT.

Considering very large perturbation bounds for AT does not provide any additional benefits and may harm robust accuracy against realistic evasion attacks. For instance, as shown in Figure 6.13, we evaluated the robustness of multiple DNN models trained with large perturbation budgets ranging from 100 to over 800 against a realistic evasion attack. The results, depicted in the plots, indicate insufficient robustness. Furthermore, the larger the

perturbation bound used during training, the worse the robustness becomes. This suggests that increasing the perturbation bound for AT does not imply that more relevant regions in the feature space are explored.

Takeaway 5: High-confidence AEs do not always matter in AT.

Our thorough investigation, particularly in Section 6.4.2.2 and Section 6.4.2.3, demonstrates that incorporating low-confidence AEs in AT can sometimes yield better adversarial robustness than using high-confidence AEs in the malware domain across different scenarios. Therefore, resolving convergence issues in the inner maximization problem of AT is of lesser immediate importance. This aligns with the findings of [210].

Takeaway 6: Using realistic evasion attacks in AT outperform unrealistic ones for hardening deep non-linear models trained on dense, low-dimensional discrete feature space.

Our observations in Figure 6.8—RAMDA (DNN, subfigures a to d)—show that DNN models (deep non-linear models) hardened with PK-Greedy and EvadeDroid (realistic evasion attacks) exhibit superior adversarial robustness compared to those hardened with PGD and JSMA (unrealistic evasion attacks) when the models trained on RAMDA (a dense, low-dimensional feature space). Moreover, as shown in Table 6.2, this effectiveness is achieved by maintaining the clean performance of hardened models similar to the vanilla DNN model trained on RAMDA. For a more detailed discussion, refer to Section 6.4.2.4.

Takeaway 7: AT with unrealistic evasion attacks is ineffective for hardening linear models trained either on a dense, low-dimensional discrete feature space or sparse, high-dimensional discrete feature space.

Table 6.2 demonstrates that the clean performance of SVM models (linear models) trained on either DREBIN (sparse, high-dimensional discrete feature space) or RAMDA (dense, low-dimensional discrete feature space) substantially drops when they are hardened with PGD or JSMA (unrealistic evasion attacks). Moreover, as shown in Figure 6.8—DREBIN (SVM, subfigures a to d)—using PGD in AT to strengthen SVM trained on DREBIN, and in Figure 6.8—RAMDA (SVM, subfigures a to d)—using both PGD and JSMA in AT to strengthen SVM trained on RAMDA outperforms PK-Greedy and EvadeDroid. However, this ultimately renders the SVM ineffective due to a considerable reduction in the clean performance of the hardened SVM models. For further details, see Section 6.4.2.4.

Takeaway 8: Unrealistic evasion attacks outperform realistic ones in hardening deep and shallow non-linear models trained on sparse, high-dimensional, discrete feature spaces.

Figure 6.8—DREBIN (DNN, subfigures a to d) and DREBIN (DT, subfigures a to d)—shows using PGD and JSMA (unrealistic evasion attacks) in AT to harden the DNN

and DT models (deep and shallow non-linear models) trained on DREBIN (sparse, high-dimensional discrete feature space) often provide better adversarial robustness than PK-Greedy and EvadeDroid (realistic evasion attacks). As shown in Table 6.2, This superiority is accompanied by maintaining the clean performance of the hardened DNN and DT models because, as discussed in Section 6.4.2.2, using unrealistic evasion attacks in AT for hardening non-linear models has a minimal impact on clean performance.

> **Takeaway 9:** Adversarial robustness is not necessarily correlated with the roughness of the decision boundaries found in the discrete feature space.

Our meticulous analysis conducted in Section 6.4.2.5 reveals that contrary to findings discussed in [209], for the malware classifiers trained on the discrete feature space, altering the decision-function roughness does not consistently affect the adversarial robustness achieved through AT.

> **Takeaway 10:** Adversarial robustness is correlated with regions in the discrete feature space that are important for both attackers and defenders.

Our thorough analysis in Section 6.4.2.5 demonstrates that the adversarial robustness of the models trained on discrete feature space can improve with an increased number of common features important in both the defending and attacking processes.

6.5 Related Work

In this section, we review related work on adversarial training with a particularly emphasis on those applied to harden malware classifiers.

Adversarial Training. Although the study by Goodfellow et al. [50] is recognized as the first to demonstrate that the inclusion of AEs during the training phase can enhance the robustness of ML models to evasion attacks, the robust optimization formulation proposed by Madry et al. [53] for AT marks a turning point in this area. Over the past few years, numerous studies have focused on robust optimization to enhance the adversarial robustness of ML models. For instance, Zhang et al. [192] demonstrated the limitations of robust optimization in ensuring robustness when test points slightly deviate from the training set distribution. Zhang et al. [192] revealed that the effectiveness of AT is closely tied to the proximity of test data to the training data manifold. Levi and Kontorovich [217] introduced an AT approach where perturbed examples from each class are regarded as distinct categories by dividing each class into *clean* and *adversarial*. Zhang et al. [218] proposed a new defense method to balance adversarial robustness and accuracy by decomposing the robust error into classification and boundary errors.

Adversarial Training for Malware Detection. AT has been regarded as the most prevalent defense mechanism for strengthening ML-based detectors. In the last few years, several studies [14, 17, 28, 30, 32, 37, 60, 206–208] have explored adversarial retraining [50] to improve the robustness of malware detectors. Most of these studies [14, 17, 28, 30, 32, 37, 60, 207, 208] have primarily focused on proposing new evasion attacks for generating AEs needed in adversarial retraining. For instance, Grosse et al. [17] adapted the JSMA [164] to

generate highly effective AEs.

One of the major concerns frequently observed in studies exploring AT is that the hardened models were not tested against realistic evasion attacks. Specifically, they did not clarify whether their attacks meet all domain constraints [9]. For example, the AEs used in [14, 17, 24, 25, 35, 38] may lack robustness to preprocessing, as preprocessing operators can potentially remove features added to the Manifest file of Android apps [9]. Additionally, bytes that are appended into non-executable areas of Portable Executable (PE) files through the attacks described in [36, 60, 208] might be discarded by preprocessing before classification [219]. It is worth noting that the adversarial attacks used in some studies, such as [31, 206], may not adequately reflect the robustness of detectors against adversaries, as they did not evaluate the adversarial robustness against some attacks conducted in PK settings, potentially allowing adversaries to create strong attacks.

Another notable concern is the lack of in-depth examination of adversarial robustness in most studies since exploring AT was not their primary focus. In recent years, only a few studies within the field of malware have dedicated their research to investigating AT. Two significant explorations have been conducted in [206] and [36]. In fact, Dyrmishi et al. [206, 210] examined the influence of domain constraints on AT by exploring realizable AEs generated in the problem space. Their findings demonstrated that in the malware domain, models hardened with unrealizable AEs exhibit less robustness against realistic evasion attacks compared to models strengthened with realizable AEs. Their observations also showed that clean performance is slightly affected after AT. Lucas et al. [36] enhanced the efficiency of various problem-space evasion attacks in generating AEs, making AT practical for raw-binary malware detectors. Furthermore, unlike the observations reported in [206], they found that using unrealistic evasion attacks in AT can provide appropriate adversarial robustness; however, it significantly degrades clean performance. In addition to these two studies, [31, 137, 153] specifically explored robust optimization for AT. Similar to [206], Bostani et al. [137] studied the impact of domain constraints on AT. However, they explored realizable AEs generated in the feature space to overcome the limitations of utilizing problem-space AEs in AT. Doan et al. [153] introduced a new adversarial learning objective based on Bayesian inference to capture the distribution of models, leading to improved robustness.

Finally, the lack of extensive investigation into the impact of influential factors on AT, especially classifiers and feature representations is another noteworthy concern in relevant studies. Most of the studies [17, 24, 25, 28–33, 35–38, 60, 153, 154, 156, 206–208] have primarily focused on using AT to reinforce a single type of classifier, typically relying on Deep Neural Networks (DNNs) for malware detection; however, it remains uncertain how effective their explored AT methods are in fortifying other types of classifiers. Furthermore, to the best of our knowledge, there has not been a thorough investigation into how different representations, such as high-dimensional and low-dimensional feature spaces, impact the effectiveness of AT in the context of malware detection.

6.6 Discussion

Our analysis demonstrates that multiple variables (e.g., classifiers, feature representations, and robust optimization settings) significantly influence the effectiveness of hardening a target ML-based malware detection through AT. For instance, the method employed to create

AEs is also crucial, as our research in this chapter indicates that for a model to exhibit robustness, there needs to be some overlap between the areas of the input space covered by the adversarial hardening approach and those exploited by an attacker. Our findings indicate that there is no perfect general formula that exists, but that each configuration must be evaluated individually and meticulously to ensure the maximum effectiveness of the hardening process. This careful evaluation is crucial to avoid configurations that could potentially harm the learning process. For instance, inappropriate choices in feature representation or learning algorithms may lead to suboptimal robustness and a drop in clean performance. Furthermore, it is essential to tailor the AT approach to the specific characteristics of datasets and the threat models. By doing so, one can enhance the model's robustness without compromising its overall performance. This highlights the importance of comprehensive and context-specific evaluations in developing AT strategies.

6.6.1 Limitations

Our research in this chapter focuses on classifiers suitable for exploring a range of models, from deep non-linear to linear models. However, further exploration is needed, as the target learning algorithms significantly impact the effectiveness of AT. Moreover, we concentrate on discrete feature representations, common in malware classification, but since altering the feature representations could greatly affect attacker capabilities and the AT process, exploring more diverse feature representations is crucial for a deeper understanding of clean performance and adversarial robustness. Expanding the exploration of AT for malware classification by adding more classifiers and feature representations can offer deeper insights, but is impractical within a single study due to its complexity. Adding just one classifier or feature representation can increase the number of models in the study conducted in this chapter from $\approx$ 1K to $\approx$ 1.5K, with training and evaluation potentially taking over a year. This highlights the trade-off between comprehensiveness and feasibility in large-scale AT experiments.

6.6.2 Future work

We underscore the real-world computational demands of large-scale AT to inspire future advancements. Our framework offers researchers a platform to explore customized adversarial settings, optimizing the balance between clean accuracy and robustness in malware detection. Given the limitations of prior studies, which often draw conclusions from narrow and varied settings, we encourage researchers to build on our framework for further exploration of the key factors highlighted in this chapter. Such investigations can help identify broad trends and establish clear benchmarks across diverse experimental, operational, and threat model contexts.

6.7 Conclusion

In this work, we present a unified framework that outlines key exploration and evaluation dimensions in AT for malware detection, along with critical factors for each. Our systematic evaluation based on this framework uncovers common evaluation pitfalls and provides key insights to improve the effectiveness of AT. Our exploration highlights how various factors influence the impact of AT, revealing a complex web of interconnected characteristics that

affect its success. Indeed, our findings suggest that a tailored approach is essential for developing robust models through AT.

6.A Experimental Settings

6.A.1 Data

To empirically analyze how the *distribution* and *volume* of data affect AT's efficacy, we use two datasets, DREBIN20 [9] and APIGraph [54], which differ in size and malware classification criteria. The DREBIN20 dataset comprises $\approx 150K$ Android applications collected between January 2016 and December 2018, while APIGraph contains $\approx 323K$ Android applications from January 2012 to December 2018. Specifically, DREBIN20 includes 135,708 *goodware* and 15,765 *malware* labeled according to criteria adapted from [220], where an app is classified as goodware if it has zero VirusTotal (VT) detections, and as malware if it has four or more VT detections. Moreover, APIGraph comprises 290,505 goodware and 32,089 malware, labeled using the criteria established in [221]: an app is considered goodware if it has zero VT detections and malware if it has 15 or more VT detections. It is noted that both DREBIN20 and APIGraph datasets are constructed to align with the composition suggested in Tesseract dataset [222], ensuring that the malware ratio mirrors the real-world prevalence of about 10% for Android malware to maintain spatial consistency, while also preserving temporal consistency by organizing samples chronologically and distributing them evenly over the years. In constructing the training and test sets, we use stratified sampling to create fair and unbiased subsets (33% for testing, 67% for training), while maintaining a 10% malware ratio in datasets. Additionally, 10% of the training set is reserved for validation. To ensure a fair and meaningful evaluation, we begin with vanilla models that perform well on clean data. It is important to note that both the datasets and the way training and test sets are constructed aim to minimize temporal and spatial biases to mitigate concept drift. Since this is an inherent challenge in malware classification, we believe concept drift should be addressed beforehand, rather than adding complexity by applying AT on top of it.

6.A.2 Feature representations

In the study conducted in this chapter, we focus on two primary data representations to examine the role of *dimensionality* and *sparsity* of data representations: DREBIN [223] (sparse, high-dimensional) and RAMDA [159] (dense, low-dimensional). Indeed, these were selected for their distinct characteristics, which make them suitable for our analysis of the impact of representation on security hardening processes. The DREBIN representation, as outlined in the work by [223], is defined by a high-dimensional, sparse feature space, consisting of approximately 1.5M features. This vast dimensionality provides a comprehensive but complex view of data, potentially capturing more nuanced aspects of behaviors and patterns. Given that the DREBIN representation is substantially high-dimensional, we select the top $10K$ most distinctive features based on recommendations from prior studies [16]. In contrast, RAMDA, as introduced by Li et al. [159], utilizes a dense, low-dimensional space with 379 features. This compact representation is beneficial for modeling as it may reduce the complexity and increase the manageability of the dataset.

The choice of these representations is significant because, as highlighted [192], the distribution of data plays a critical role in AT. Smaller dimensions, such as those used in RAMDA, can potentially improve the fidelity of AT samples to the true data distribution.

6.A.3 Classifiers

We employ three models: DNN as a deep non-linear classifier, DT as a shallow non-linear classifier, and linear SVM as a linear classifier to investigate how different learning algorithms are influenced by AT. In fact, since these classifiers range from non-linear to linear, they are well-suited for assessing the *model flexibility* factor highlighted in the proposed framework.

6.A.4 Threat Model

In order to conduct a thorough investigation, we consider a comprehensive threat modeling scenario that examines both defender and attacker capabilities. As an attacker, our goal is to fool the classifier and evade it successfully, while as a defender, we want to make our model as robust as possible against multiple threat actors. Our analysis focuses on four key factors:
(a) Adversarial Confidence: Adversarial confidence refers to the model's certainty in classifying an input program as either malicious or benign, playing a key role in both the success of attacks and the robustness of the model. Attackers aim to generate adversarial malware that can bypass detection with high confidence, while defenders work to reduce this confidence and mitigate the associated threats.
(b) Perturbation Bound: This capability defines the maximum allowable perturbation that can be applied to a target sample. For example, given a target binary vector and a perturbation bound of 5, it means that in order to craft an AE, we set a modification upper bound of 5 features. This parameter is crucial for both attackers and defenders as it directly impacts the confidence and feasibility of the generated samples. It is important to highlight that, from the attacker's perspective, a lower perturbation bound restricts the number of features that can be modified, which may hinder the success of evasion. On the other hand, higher bounds can increase the chances of successful evasion but may also raise the risk of detection through behavioral analysis or manual inspection, as well as the risk of altering the functionality of the malware program.
(c) Adversarial Fraction: This capability indicates the proportion of AEs used during the training phase. For instance, if the adversarial fraction is set to 5%, it implies that only 5% of the available malware samples in each batch are used for AT. While this parameter is primarily set by the defender during training, it indirectly reflects the attacker's influence. A lower adversarial fraction results in less exposure to adversarial behavior, potentially leaving the model more vulnerable. From the attacker's standpoint, this affects their success rate, as models trained with limited exposure to AEs may generalize poorly to unseen attacks.
(d) Domain Constraints: In real-world scenarios, attackers must ensure that any modifications preserve the malicious functionality, cannot be easily removed, and appear plausible. These constraints significantly restrict the space of allowable perturbations, particularly in realistic evasion attacks. While defenders may enforce or relax such restrictions during training—leading to strategies that may or may not reflect actual attack behavior—realistic evasion attacks, from the attacker's perspective, are limited to feasible regions. These regions are typically shaped by the domain constraints that defenders attempt to capture or approximate.

By examining the capabilities of threat models in terms of adversarial confidence, perturbation bounds, adversarial fractions, and domain constraints, we can comprehensively assess the dynamics between attackers and defenders. This approach not only helps in understanding how attackers can optimize their strategies within given constraints but also

provides insights into how defenders can effectively adjust their training processes to build more resilient models. When generating AEs either to strengthen or deceive malware classifiers, the following factors are also considered for evasion attacks:

Attacker Knowledge Assumptions: To deeply explore a range of attack scenarios, we consider both Perfect Knowledge (PK) and Zero Knowledge (ZK) threat models. In the PK setting, the attacker has full access to the target model, including its architecture, parameters, data, and the feature space it operates on. In contrast, the ZK setting only allows black-box querying of the model's output.

Attack Strategies: In evaluating hardened malware classifiers, our primary focus is on realistic, problem-space evasion attacks. For PK scenarios, we utilize the PK-Greedy attack from Pierazzi et al. [9], which can also be considered an adaptive attack. This is because PK-Greedy not only functions in white-box settings but also adapts to the target model by identifying the most influential benign features specified by the hardened classifier. The attack dynamically adjusts to the model's characteristics, adapting to what is most relevant for evading detection. Additionally, PK-Greedy's method of sorting problem-space transformations and selecting the most effective one in each iteration via a greedy search further reflects its adaptive nature. For ZK scenarios, we employ EvadeDroid [84], which is a realistic, decision-based problem-space attack. Both PK-Greedy and EvadeDroid are crafted to produce realizable AEs by enforcing domain constraints, ensuring that the generated malware remains both valid and functional, which makes them particularly suitable for real-world adversarial evaluations. In some of our experiments, PK-Greedy and EvadeDroid are also used in AT to strengthen malware detectors. Additionally, we incorporate PGD [53] and JSMA [164] in AT—both unrealistic feature-space evasion attacks that use gradients to directly generate AEs in the feature space.

6.A.5 Computational Resources

All our experiments were conducted on a dedicated instance equipped with an NVIDIA A100 80GB GPU, a 32-core AMD EPYC Milan processor @2.6 GHz, 128GB of RAM, and a 2.5TB SSD.

6.B Implementation Details

In the study conducted in this chapter, we employed various learning algorithms implemented in PyTorch: linear SVM, DT, and DNN. To ensure complete control over the training process, we approximated the linear SVM as a single-layer neural network using PyTorch [224]. The SVM model was evaluated using the `LinearSVC` class from scikit-learn [225], which utilizes the LIBLINEAR library [226], to ensure our implementation work well. For the DREBIN representation, we set the hyperparameter $C = 1$ [223], while for RAMDA, our preliminary evaluations suggested $C = 4$ provides better performance. Moreover, for DT, we implemented the approach described in [227] and tuned it using Optuna [228], adjusting the following hyperparameters.

```
max_depth': 5,
'output_dim': 2,
'momentum': 0.53,
'lmbda':0.47,
```

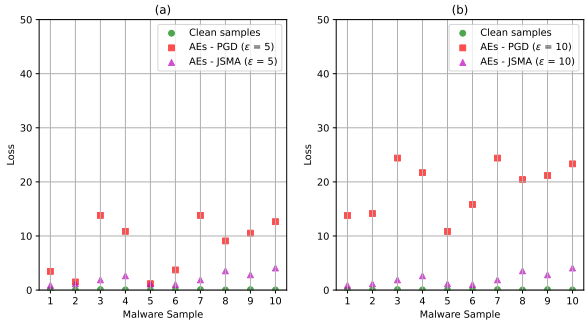


Figure 6.11: An example demonstrating the confidence level of different sample types (i.e., clean malware and adversarial malware samples) in terms of loss. A larger loss indicates greater confidence in misleading classifiers.

```
'learning_rate': 0.12,
'weight_decay':  5e-4,
```

Our implementation of the DNN is based on the Multilayer Perceptron described in [17] for malware detection.

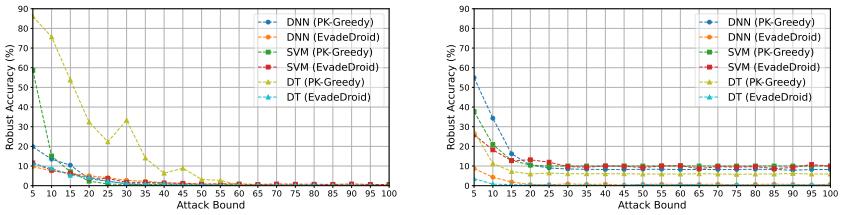
Moreover, we implemented PGD and JSMA, with the former adapted for the Android malware domain according to [24], and the latter directly adapted for use in our framework. Lastly, for PK-Greedy and EvadeDroid, we utilized the codes shared in their respective studies—[9] and [84], respectively.

6.C Confidence of AEs

To ascertain whether the feature-space attacks can generate AEs with varying levels of confidence, we conduct a preliminary evaluation of their performance against a vanilla DNN trained on the DREBIN representation. Drawing inspiration from [190], we evaluated the loss (i.e., prediction error) of the vanilla DNN when classifying AEs generated by PGD or JSMA targeting the vanilla DNN. It is important to note that a higher loss corresponds to a higher evasion confidence. Figure 6.11 illustrates the confidence levels of 10 randomly selected malware samples from the DREBIN test set, comprising both clean and adversarial examples. It is evident that AEs inherently exhibit higher confidence than clean samples, with those generated by PGD showing higher confidence than those generated by JSMA. Furthermore, as ϵ increases, the confidence of PGD-generated AEs rises, whereas it remains relatively stable or changes only slightly for JSMA.

6.D Robustness Evaluation

To assess adversarial robustness, we evaluate the models' resistance to bounded PK-Greedy and EvadeDroid attacks, ensuring that their attack bounds match those used in the evasion attacks during AT. This is because models strengthened under a particular ϵ during AT are anticipated to offer resilience within that bound, not necessarily beyond it. Therefore, we limit the maximum alterations feasible by realistic evasion attacks to ϵ. Additionally, it is imperative to confirm that the observed adversarial robustness against evasion attacks is



(a) DREBIN (DREBIN20) (b) RAMDA (DREBIN20)

Figure 6.12: Robust accuracy of various vanilla models trained on DREBIN and RAMDA against PK-Greedy and EvadeDroid with different attack bounds.

attributable to AT, rather than stemming from the limitation imposed by the attack bounds of evasion attacks. To this end, we need to quantify the improvement in the adversarial robustness of vanilla models when an attack bound is applied to evasion attacks. Therefore, we first meticulously evaluate the robustness of the vanilla model against unbound PK-Greedy and EvadeDroid to understand the inherent adversarial robustness of these baseline malware detectors, even when attacks do not adhere to a specific attack bound. In an unbounded attack, adversaries can potentially apply any combination of problem-space transformations to convert a malware app into an adversarial one, irrespective of how many features are changed in the feature space. Table 6.4 shows different models have some level of robustness against realistic evasion attacks. Especially, the models trained on RAMDA demonstrate significantly greater adversarial robustness compared to those trained on DREBIN, supporting the authors' claim in [14] that their representation can enhance robust malware detection. Next, we consider an attack bound ϵ for PK-Greedy and EvadeDroid to observe how it limits these attacks in fooling target baseline malware detectors. Figure 6.12 shows that a lower attack bound results in a higher limitation on the success of PK-Greedy and EvadeDroid.

6.E Large Perturbation Bound

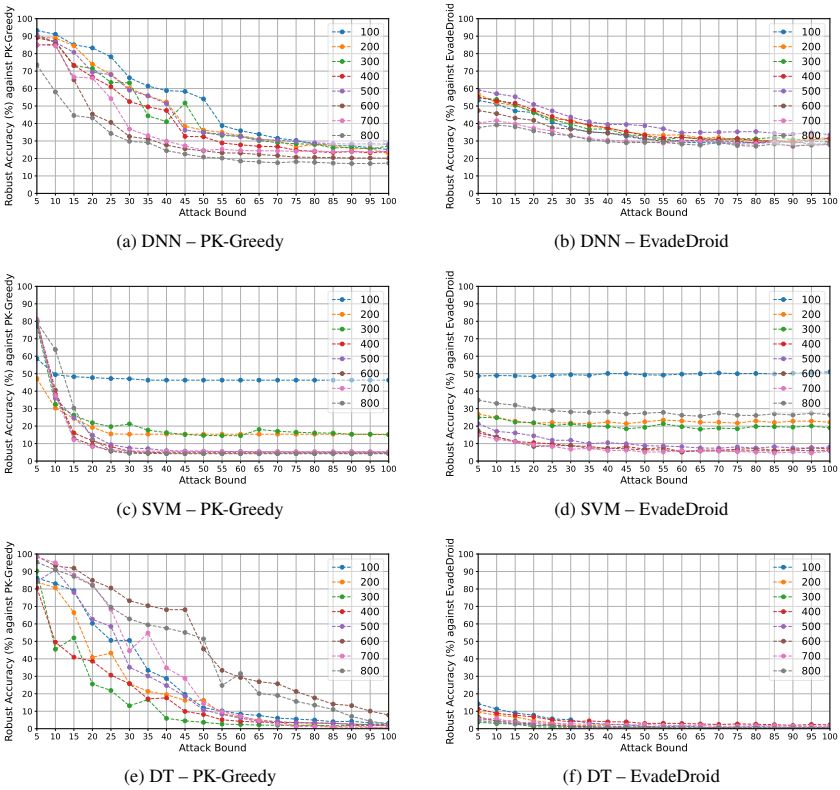
In the setting used in Section 6.4.2.2, the attacker's capabilities remain unchanged, while we consider a defender that increases the perturbation bound. The motivation is that since DREBIN is a high-dimensional feature representation, the previously considered range of perturbation bounds might be inadequate to uncover vulnerable areas.

Here, we assess the hardening of DNN, SVM, and DT models trained on DREBIN with

Table 6.4: Robust accuracy of different vanilla models against unbounded PK-Greedy and EvadeDroid.

Model	DREBIN		RAMDA	
	PK-Greedy	EvadeDroid	PK-Greedy	EvadeDroid
DNN	0.0%	0.2%	8.3%	0.6%
SVM	0.0%	0.7%	10.0%	9.4%
DT	0.0%	0.0%	6.2%	0.4%

PGD, considering large perturbation bounds that may help uncover more vulnerable areas. Specifically, the perturbation bound varies from 100 to 800 in increments of 100 to examine whether very large bounds can uncover more blind spots and enhance the effectiveness of AT. Our results, shown in Figure 6.13, demonstrate that the learning algorithm plays a vital role in this phenomenon, as shown by the differing performance of the models when exposed to attacks with smaller perturbations. We observe an evident contrast in behavior between the models: linear SVM do not benefit from larger perturbations against both EvadeDroid and Pk-Greedy for any attack bound. DNN has a slight improvement against EvadeDroid, while DT exhibits an advantage against PK-Greedy, especially for lower attack bounds, when using 600 as the perturbation bound for PGD. However, results for large perturbation bound attacks (ϵ >90) are nearly consistent across all models, indicating that this approach has a negligible or even negative effect, as exemplified by the linear SVM in Figure 6.13.



(a) DNN – PK-Greedy

(b) DNN – EvadeDroid

(c) SVM – PK-Greedy

(d) SVM – EvadeDroid

(e) DT – PK-Greedy

(f) DT – EvadeDroid

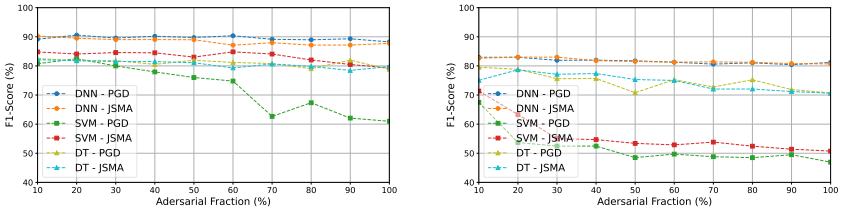
Figure 6.13: Relative robust accuracy of hardened DNN, SVM, and DT on the DREBIN feature representation of the DREBIN20 dataset against PK-Greedy and EvadeDroid. Each line represents the same model trained using the specified perturbation bound for generating AEs during adversarial training.

6.F Clean Performance Considering Different Adversarial Fractions

This section includes the evaluation of experiments that focus on evaluating the clean performance of various models trained on DREBIN and RAMDA representations of both DREBIN20 and APIGraph datasets in terms of F1 score as shown in Figure 6.14. The models are strengthened using either PGD or JSMA with different fractions of AEs.

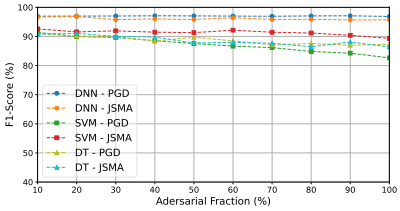
6.G Robust Performance Considering Different Adversarial Fractions

This section includes the evaluation of experiments that focus on exploring the robust accuracy of various models trained on DREBIN and RAMDA representations of both DREBIN20 and APIGraph datasets in terms of relative robust accuracy as shown in Figure 6.15. The models are strengthened using either PGD or JSMA with different rates of AEs during the training.

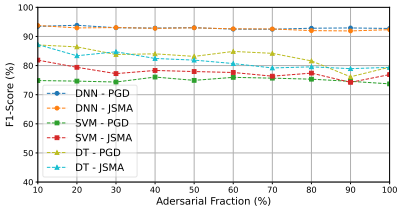


(a) DREBIN (DREBIN20)

(b) RAMDA (DREBIN20)



(c) DREBIN (APIGraph)



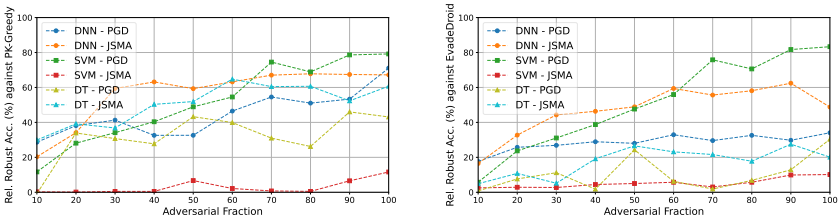
(d) RAMDA (APIGraph)

Figure 6.14: Clean performance of models trained on DREBIN and RAMDA representations of DREBIN20 and APIGraph datasets in terms of F1 score hardened with different fractions of AEs.

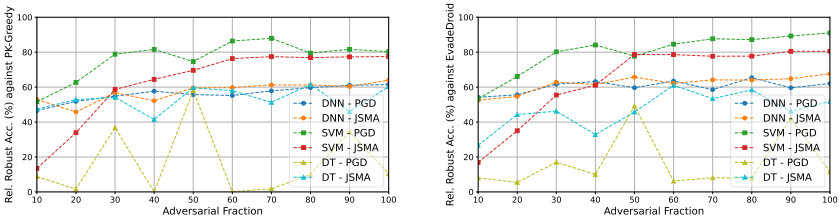
6.H Challenge of Exhaustive Exploration

Although we aimed to systematically evaluate the impact of various AT settings, conducting a comprehensive exploration across all experimental factors would require training a prohibitive number of models. Specifically, the total number of configurations is derived from the following combinations:

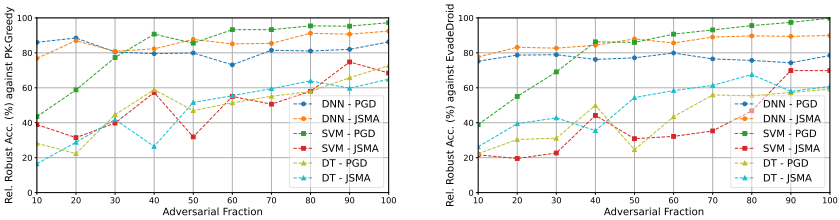
- **Datasets**: 2 (DREBIN20, APIGraph)
- **Feature Representations**: 2 (DREBIN, RAMDA)



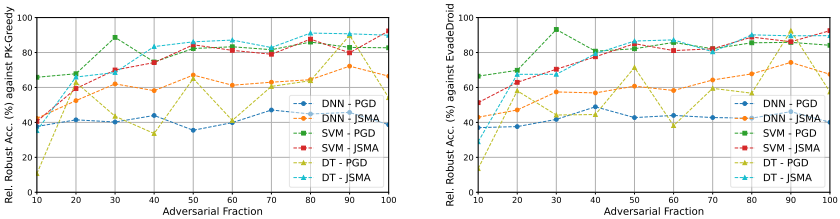
(a) DREBIN (DREBIN20) – PK-Greedy (b) DREBIN (DREBIN20) – EvadeDroid



(c) RAMDA (DREBIN20) – PK-Greedy (d) RAMDA (DREBIN20) – EvadeDroid



(e) DREBIN (APIGraph) – PK-Greedy (f) DREBIN (APIGraph) – EvadeDroid



(g) RAMDA (APIGraph) – PK-Greedy (h) RAMDA (APIGraph) – EvadeDroid

Figure 6.15: Relative robust accuracy gained from AT on hardened models trained on (a) and (b) DREBIN representation and (c) and (d) RAMDA representation of the DREBIN20 dataset, and (e) and (f) DREBIN representation and (g) and (h) RAMDA representation of the APIGraph dataset against PK-Greedy and EvadeDroid considering different fraction of AEs during training.

- **Classifiers**: 3 (DNN, linear SVM, DT)
- **Perturbation Bounds**: 28 (20 for regular perturbations + 8 for very large perturbations)
- **Adversarial Fractions**: 10
- **Adversarial Confidence Levels**: 2 (low confidence (JSMA) and high confidence (PGD))

- **Domain Constraints**: 4 (PK-Greedy, EvadeDroid, PGD, JSMA)

By systematically varying each factor independently, the total number of potential model configurations follows:

$$2 \times 2 \times 3 \times 28 \times 10 \times 2 \times 4 = 26,880$$

This calculation represents the full scope of model variations required for an exhaustive evaluation of AT across all dimensions. Given past training experience—where 1K+ models required over six months—evaluating 26,880 models would be computationally infeasible, requiring several years. Furthermore, expanding the current experimental setup by introducing additional variables into each designed experiment, such as considering various adversarial fractions in Section 6.4.2.2, would significantly increase complexity. This would make the problem analogous to an NP-hard problem, rendering it intractable within practical time constraints.

6.I Overview of Results

To simplify the review of the empirical results shown across various plots, we offer a high-level summary of the findings from our evaluations, as presented in Table 6.5.

Table 6.5: Summary of Findings.

Feature Space	Attack used in AT	Attacker's Knowledge	Type of AEs in AT	Variable	Linear Model		Non-linear Model	
					Clean Perf.	Robust Acc.	Clean Perf.	Robust Acc.
Discrete high-dim. sparse space	Feature-space, Unrealistic	PK	High-Conf. AEs	Increasing Pert. Bound	- More loss	- More gain	- No impact (deep) - Slight loss (shallow)	- No trend
				Very Large Bound	Unexamined	No effect or worse	Unexamined	No effect or worse
				Increasing AE Fraction	- More loss	- More gain	- No impact (deep) - Slight loss (shallow)	- More gain
Discrete high-dim. sparse space	Feature-space, Unrealistic	PK	Low-Conf. AEs	Increasing Pert. Bound	- Slight loss	- No trend	- No impact (deep) - Slight loss (shallow)	- No trend - Higher robust acc. compared to the high-conf. AEs setting
				Very Large Bound	Unexamined	Unexamined	Unexamined	Unexamined
				Increasing AE Fraction	Sligh loss	- More gain, but lower than the high-conf. AEs setting	- Slight loss	- More gain
Discrete low-dim. dense space	Feature-space, Unrealistic	PK	High-Conf. AEs	Increasing Pert. Bound	-Varied loss	- No trend	- No impact (deep) - Varied loss (shallow).	- No trend - Higher robust acc. compared to the high-dim. space setting (shallow)
				Very Large Bound	Unexamined	Unexamined	Unexamined	Unexamined
				Increasing AE Fraction	- More loss	- More gain	- No impact (deep) - More loss (shallow)	- More gain
Discrete low-dim. dense space	Feature-space, Unrealistic	PK	Low-Conf. AEs	Increasing Pert. Bound	- More loss - Higher clean perf. compared to the high-conf. AEs setting	- More gain	- No impact (deep) - Varied loss (shallow)	- No trend - Higher robust acc. compared to the high-conf. AEs setting (deep)
				Very Large Bound	Unexamined	Unexamined	Unexamined	Unexamined
				Increasing AE Fraction	- More loss - Higher clean perf. compared to the high-conf. AEs setting	- More gain	- No impact (deep) - More loss (shallow)	- More gain - Higher robust acc. compared to the high-conf. AEs setting
Discrete high-dim. sparse space	Problem-space, Realistic	PK	Realizable AEs	Forcing Domain Constraints	- No impact	- Low robust acc. (only against similar attack)	- No impact	- High robust acc. against similar attack (deep) - Very low robust acc. (shallow)
Discrete high-dim. sparse space	Problem-space, Realistic	ZK	Realizable AEs	Forcing Domain Constraints	- No impact	- Low robust acc. (only against similar attack)	- No impact	- Low robust acc. against similar attack, and very low against dissimilar attack (deep) - Very low robust acc. (shallow)
Discrete low-dim. dense space	Problem-space, Realistic	PK	Realizable AEs	Forcing Domain Constraints	- No impact	- Moderate robust acc.	- No impact	- High robust acc. against similar attack (deep) - High robust acc. (shallow)
Discrete low-dim. dense space	Problem-space, Realistic	ZK	Realizable AEs	Forcing Domain Constraints	- No impact	- Moderate robust acc.	- No impact	- Very high robust acc. (deep) - Moderate robust acc. against similar attack and high robust acc. against dissimilar attack (shallow)

7

Conclusions and Outlook

Nowadays, ML has become a ubiquitous tool, offering promising opportunities across various applications, especially in the context of malware detection [230]. However, safeguarding ML systems against evasion attacks remains a major concern for enhancing the adversarial robustness of malware classifiers. While previous studies have made notable efforts to improve the robustness of ML-based malware detection, their practical effectiveness is often questionable due to a lack of realism, such as overestimating attackers' knowledge or overlooking feasible vulnerabilities in the decision space of malware classifiers. In this dissertation, we have rethought the security of malware classifiers against evasion attacks by exploring realism concerns on both the attacker and defender sides, with the goal of assessing the current state and paving the way toward resolving this critical challenge.

In this chapter, we first summarized our findings in Sections 7.1 and 7.2, based on our explorations of ML systems for malware detection, emphasizing realism, which is essential in developing realistic evasion attacks and reliable defenses. Then, in Section 7.3, we present an overview that highlights new research directions for enhancing the robustness of malware classifiers.

7.1 Realistic Threat Models

Chapter 3 has provided an in-depth exploration to address Q1 defined in Chapter 1:

> **Q1: What threat model is truly practical for evasion attacks targeting malware classifiers in real-world scenarios?**

In this chapter, we have taken into account realism concerns from the perspective of attackers by introducing *EvadeDroid*, a practical evasion attack targeting black-box Android malware detectors. Our findings highlight several key requirements for advancing evasion attacks based on realistic threat models in the malware domain.

1. **Develop Attacks That Are Agnostic to the Target Model.** While evasion attacks in a perfect-knowledge setting are useful for robustness evaluation as a worst-case scenario, it

Section 7.3 of this chapter is based on the accepted paper: H. Bostani and V. Moonsamy, Beyond Learning Algorithms: The Crucial Role of Data in Robust Malware Detection, IEEE Security & Privacy, 2025 [229]. Along with revising the second paragraph of Section 7.3 and incorporating a new paragraph at the beginning of Section 7.3.3, Subsections 7.3.1 and 7.3.2 have been added to broaden the outlook beyond the original paper.

is equally important to develop attacks that are agnostic to the target model. This ensures practical assessments of malware classifiers in real-world scenarios, where detectors operate as black boxes, often returning only hard labels.

2. **Address the Challenge of Inverse Feature Mapping.** Adversaries must generate adversarial malware in the problem space, even if perturbations are crafted in the feature space. Reconstructing malware based on feature representations is challenging since the feature mapping from the problem space to the feature space is not invertible. As confirmed by EvadeDroid, this issue can be resolved by directly identifying adversarial perturbations in the problem space, eliminating the need to identify them in the feature space.

3. **Generate Realizable Adversarial Malware.** Adversarial malware must not only be functional programs capable of compromising victims' machines but also appear as valid programs and ensure their adversarial payloads–such as dead or rubbish code injected by adversaries to deceive malware classifiers–are not excluded during preprocessing, a common step in ML pipelines. Our findings show that adversarial payloads injected into obfuscated `FALSE` conditions remain inactive at runtime while staying undetectable during preprocessing, ensuring they cannot be removed. Moreover, using snippets of benign code as adversarial payloads ensures adversarial malware appears as valid software even when inspected by human programmers.

4. **Ensure Low Evasion Costs.** Adversaries need to optimize the query efficiency of their methods, as excessive query requirements are not economical and, from a practical perspective, might raise suspicion and lead the target detector to impose restrictions or block further queries. Our investigation shows that utilizing existing code snippets–well-crafted problem-space transformations extracted from benign programs at API calls entry points–plays an essential role in making the attack query-efficient.

7.2 Reliable Defenses

To move towards reliable defenses, Chapters 4 to 6 have investigated various aspects of defending against realistic evasion attacks:

7.2.1 Identifying Vulnerable Regions

Our findings in Chapter 4 suggest that practical defense mechanisms must efficiently and effectively identify feasible vulnerable regions while also focusing on these regions without bias toward specific areas. Specifically, Chapter 4 has offered a comprehensive analysis to answer Q2 defined in Chapter 1:

> **Q2: How can vulnerable regions in malware classifiers be efficiently identified to defend against realistic evasion attacks?**

In Chapter 4, we have highlighted that identifying feasible regions vulnerable to realistic attacks requires generating realizable adversarial malware. The chapter has empirically demonstrated that generating adversarial malware directly in the feature space, rather than

the problem space, is more efficient. Additionally, it provides defenders with a greater opportunity to protect against unknown attacks that exploit new vulnerable regions. This is because using known problem-space transformations to craft realizable adversarial malware biases defenders toward specific regions, whereas directly exploring the feature space to identify vulnerabilities avoids such limitations.

7.2.2 Reducing Spurious Correlations

Our findings suggest that effective defenses should address spurious correlations to ensure the practicality of malware classifiers in real-world scenarios. Specifically, our exploration in Chapter 5 has shown that malware classifiers have low generalizability, particularly against adversarial malware, because they tend to learn misleading patterns unrelated to malicious functionalities. In this chapter, we have provided an extensive analysis of spurious correlations in the malware domain to address Q3 defined in Chapter 1:

> **Q3: How can the impact of spurious correlations be reduced to secure malware classifiers against realistic evasion attacks?**

The chapter had discussed the significance of feature representations in minimizing the risk of learning spurious correlations, which could significantly impair the generalizability of malware classifiers. We have indeed highlighted that ML models could be guided to learn genuine patterns rather than misleading ones by ensuring that the features, representing the dimensions of the feature space, are semantically aligned with the programs' functionalities.

7.2.3 Effective Adversarial Training

In Chapter 6, we have demonstrated that considering all key factors in both the hardening and evaluation processes is crucial for understanding the effectiveness of Adversarial Training (AT). This chapter has specifically conducted comprehensive evaluations to address Q4 defined in Chapter 1:

> **Q4: What are the key factors influencing adversarial training, and how do they affect its effectiveness in malware detection?**

Our findings highlight that data, feature representations, classifiers, and robust optimization settings are key exploration dimensions that must be examined to identify and understand the essential factors influencing the performance of AT in hardening malware classifiers. Particularly, we have identified different influential factors in each key dimension as follows:

1. **Data:** The *distribution* and *volume* of training data can significantly impact the outcomes of AT.

2. **Feature Representations:** *Dimensionality*, *sparsity*, and the *type* of feature representation are influential factors in achieving robust models.

3. **Classifiers:** *Model flexibility* is important, as different learning algorithms exhibit varying resilience to adversarial attacks.

4. **Robust Optimization Settings:** In addition to the *perturbation bound* used to generate adversarial examples (AEs) for AT, *adversarial confidence* (i.e., the confidence of the AEs), *adversarial fraction* (i.e., the number of AEs used in AT), and *domain constraints*, ensuring the realizability of the AEs, are crucial configurations for strengthening malware classifiers.

It is worth to note that each of these factors plays a distinct role in the success of AT. However, it is essential to recognize their intertwined roles, rather than focusing solely on their individual contributions. Additionally, a fair evaluation of adversarial robustness is vital, necessitating the use of realistic evasion attacks based on diverse threat models, along with employing reliable metrics to assess the performance of malware classifiers.

7.3 Outlook

In today's digital age, malware is a formidable threat, compromising countless users daily. According to AV-TEST's findings, over 450K new malicious programs and potentially unwanted applications are detected each day [231]. This staggering number highlights the immense threat to our online safety. So, how do we keep up with these evolving dangers? This is where ML systems come into play. Unlike traditional methods, ML systems can learn from data and adapt to new attack strategies, significantly enhancing their effectiveness against sophisticated malware. But here is the catch: making sure ML-based malware detection systems are reliable is no small feat. For these systems to be truly dependable, they need to follow the core principles of trustworthy ML, including robustness, fairness, explainability, privacy, and transparency [232]. Among these, robustness is particularly crucial in the context of malware detection due to the adversarial nature of malware. For instance, Trojan horses are a common type of malware inherently designed to appear as legitimate programs, thereby deceiving detection systems. Ensuring robustness becomes even more critical when dealing with evasion attacks.

Although significant progress has been made in improving the adversarial robustness of malware classifiers, these systems remain vulnerable to adversarial threats like evasion attacks which aim to compromise classifiers by manipulating unseen data during the inference phase. This thesis has made significant efforts to improve the security of ML-based malware detection systems, specifically those used in static analysis, against evasion attacks, with an emphasis on realism–a crucial factor in real-world scenarios. However, ML-based dynamic analysis [233] is also widely used for malware detection alongside static analysis, as practical defenses typically incorporate multiple layers of protection [234]. Therefore, an important research direction is to investigate the robustness of these systems against evasion attacks that manipulate dynamic features extracted during program execution. Moreover, we believe that further investigation, particularly in the following directions, is necessary to effectively secure ML-based malware detection systems.

7.3.1 The Impact of Malware Variability in Evasion Attacks

The diversity of malware threats could have significant implications for evasion attacks. Different malware types and families may not only vary in their objectives but also in how adversaries craft evasion techniques to bypass detection systems. For instance, ransomware typically tightly couples encryption mechanisms with its payload, which may make it harder

to apply adversarial perturbations by complicating the modification of critical components that should be exploited for evasion. In contrast, spyware, which primarily focuses on data collection, may be more susceptible to adversarial modifications. As it relies on API calls to gather user information, injecting benign permissions or APIs could enable adversarial transformations while maintaining functionality.

While this thesis primarily focuses on exploring adversarial robustness in malware detection, understanding how evasion techniques interact with different malware threat models is an important aspect of ensuring the practicality of defenses. This broader perspective suggests a need for further research into tailoring adversarial defenses to account for the variability in malware features, evasion capabilities, and detection methods. Therefore, a promising avenue for future research is to explore how different malware types or families influence the complexity of evasion attacks in generating adversarial malware.

7.3.2 Detecting Malware in Public Repositories

Malware developers continually devise innovative and sophisticated methods to infiltrate victim machines and execute malicious code. One of the more recent tactics for distributing malware involves exploiting public code repositories, such as GitHub [235]. In this approach, adversaries create malicious repositories by embedding harmful source code within legitimate codebases. These malicious repositories are often designed to closely resemble legitimate ones, making it challenging to differentiate between the two manually. When such repositories are cloned or their code is integrated into other projects, the malware can spread, potentially leading to data breaches, system compromises, and even widespread exploitation of vulnerable software systems [236].

The increasing prevalence of this attack vector has led to the application of ML techniques to detect malicious repositories [237]. Although the use of ML for identifying malicious code in repositories is still in its infancy, it holds considerable promise for automating and scaling detection efforts. However, as with any technology, ML models are susceptible to evasion attacks. Given that the application of ML in this domain is still evolving, it seems essential to proactively explore methods to safeguard these systems against potential adversarial manipulations. A compelling direction for future research is to investigate the security of ML-based detection systems for malicious repositories. This would involve not only improving detection accuracy but also enhancing the robustness of these systems to evade attacks, ensuring their effectiveness in identifying malicious code even in the presence of sophisticated adversarial techniques.

7.3.3 Data's Role in Robust Malware Detection

Generally, strengthening the robustness of malware detection systems involves refining learning algorithms or enhancing the quality of the training data. Recent research has largely focused on improving algorithms, but a crucial factor in the generalizability of ML models–their ability to perform well on unseen data–depends on the quality of the data itself. ML models often excel when working with in-distribution data; but, they struggle when confronted with out-of-distribution examples, especially adversarial ones. This issue is particularly evident in malware detection, where the threat landscape is constantly shifting.

7.3.3.1 The Art of Building Robust Malware Classifiers

Think of cybersecurity as a battlefield where we must constantly adapt to new threats. Building robust malware classifiers is no different. Now, imagine a training dataset with only one malicious and one benign sample–hardly enough to prepare for the fight. In fact, training or adversarial training [148] an ML model on such minimal data leads to underfitting, making it unable to distinguish between malware and benign software. This results in poor performance on unseen and adversarial malware, as the model has not learned the broader characteristics that define malicious and benign software. As we add more samples to our training set, the model gets better at recognizing a wider range of features and variations. This helps the model accurately classify new, unseen malware and withstand adversarial attempts to deceive it. However, this process must be balanced with the risk of overfitting, where the model memorizes the training data instead of learning from it, leading to poor performance on new data.

But the question remains: Does simply increasing the number of samples in the training set consistently improve both accuracy and robustness? While practitioners might instinctively answer "Yes", especially when using deep neural networks for malware detection, the reality is far more nuanced. Developing robust malware classifiers requires a delicate balance between data quantity and quality to build models capable of handling the ever-evolving threats of malware.

7.3.3.2 Why Should We Care About Data?

Data is the fuel of the digital era. It is what shapes our knowledge and powers our technologies. Data is indeed the foundation upon which information is built, and information, in turn, forms the basis of knowledge [238]. Errors in data can lead to flawed knowledge [238]. Just like a car needs high-quality fuel to run smoothly, ML models need high-quality data to perform effectively. However, not all data is created equal. Noisy, biased, or inaccurately labeled samples can cause serious complications, especially for ML systems used in malware detection. These data errors can lead to inaccurate models, making them unreliable when it matters most, such as detecting malware in real-world cybersecurity scenarios.

Contrary to popular belief, more data is not always better. Studies in both supervised and unsupervised learning [239, 240] have demonstrated that merely adding more data can sometimes hurt performance, especially when the additional data lacks quality. For instance, Loog et al., [239] discusses how empirical risk minimization, a fundamental aspect of statistical learning theory aiming to minimize average loss across a dataset, does not always benefit from larger datasets. Likewise, Loog et al., [240] examines unsupervised learning methods such as k-means clustering, which can fail to improve with additional data when that data is noisy or misrepresented. These insights highlight a crucial lesson: Low-quality data can undermine the performance of malware classifiers, especially their robustness. So, how does data quality shape model robustness? Understanding this relationship is essential for developing resilient ML systems.

7.3.3.3 Robustness Is Tied to Data Quality

In the fight against malware, the quality of the data used to train ML models is crucial. High-quality training data enable models to learn robust features–genuine, meaningful patterns relevant to detecting malware. When a model is trained on robust features, it performs well and generalizes effectively. However, models often end up learning non-robust

features that are highly predictive but might seem unrelated or illogical to humans [198]. This issue can arise from low-quality data, such as biased or incomplete datasets.

To illustrate the significance of data quality in providing resilient models, consider an image detection model designed to classify seagulls. If the training set is biased, containing mostly images of seagulls with blue backgrounds (e.g., the sea), the model might incorrectly associate the blue background–non-robust feature–with the presence of seagulls. This makes it vulnerable to adversarial attacks, as an adversary could manipulate the background without altering the actual seagull. While such changes are imperceptible to humans, who still recognize the seagull, the model would fail. Conversely, if the model learns robust features–like the seagull's beak–it becomes more resilient, as adversaries would need to alter these critical features, which are harder to manipulate without violating perceptual constraints.

The same principle applies to malware detection. Biased datasets can have a similar detrimental impact. Let us consider the following scenario where malware apps are frequently collected from Chinese markets, while benign apps come from various markets, such as Google Play. Since there are so many of these samples, the model starts to think that features derived from market data are indicative of malicious behavior. In fact, an ML classifier might mistakenly associate this market data with malicious behavior, even though there is no real connection. This misclassification happens because the model is picking up on irrelevant features due to the biased training dataset.

Improving learning algorithms through techniques like regularization can help mitigate these issues, but enhancing data quality offers a more fundamental solution. One effective approach is subsampling. For instance, instead of using all malware samples from a certain market–which typically dominates the training dataset–we can randomly select a smaller, more representative subset. This reduces the over-representation of certain features, allowing the model to focus on the actual patterns that distinguish malware from benign software. This process, known as subsampling from the majority class in the training set, helps mitigate errors that arise when models are over-parameterized [241].

The subsampling approach appears promising for selecting key samples that capture the core characteristics of the dataset. However, this brings us to a crucial question: How can we ensure that training datasets for building ML systems are composed of high-quality, representative samples? Addressing this question is particularly important in malware detection. High-quality, representative datasets allow the models to learn robust features associated with genuine patterns. Learning robust features increases the generalizability of malware classifiers, enabling them to perform reliably in real-world scenarios where the variety and sophistication of malware are constantly evolving.

7.3.3.4 Coresets and Adversarial Robustness

When it comes to ML systems, the quality of our training data is crucial. Think of it like cooking. Using fresh, high-quality ingredients leads to a delicious meal, while using subpar ingredients can ruin the dish. Similarly, in ML, not all data samples are created equal. Some are the fresh, high-quality "ingredients" that capture the true essence of the dataset, while others can introduce noise and redundancy, making it hard for the model to generalize effectively [242]. So, how do we ensure our ML models get the best "ingredients"? The best we can do is use a subsampling method, and we believe coreset construction

techniques [243] are among the most promising subsampling methods for retaining only high-quality, representative samples in the training dataset. In fact, instead of training on the entire dataset, which can be overwhelming and inefficient, coresets allow us to focus on smaller, more representative subsets of data. This approach not only speeds up training but also achieves performance similar to training our ML model on the entire dataset, as the coreset preserves the key characteristics of the training dataset. Figure 7.1 (a)[1] is an example showing that models trained on a coreset can achieve performance comparable to those trained on the full dataset. In fact, by using a coreset construction method, we can cut down on training time while still hitting the same optimal performance.

But here is where it gets even better. Coresets are not just about efficiency. They also can enhance the model's robustness against adversarial attacks [245]. Think of it like training a security guard. By focusing on the most representative training scenarios, the guard becomes better at spotting real threats. Figure 7.1 (b) highlights how training on high-quality samples identified by the CRAIG method [244], a coreset construction method, leads to strong generalization performance, often surpassing models trained on the full dataset.

In practical terms, focusing on the most diverse and representative samples helps reduce noise, redundancy, and outliers–elements that can make a model vulnerable to adversarial attacks. This is particularly vital in cybersecurity applications like malware detection, where adversaries constantly evolve their techniques. By capturing both malicious and benign behaviors more effectively, coresets make our ML models stronger and more reliable. Moreover, coresets act as a form of regularization, guiding models toward simpler, more robust decision-making. This does not just improve generalizability but also reduces the risk of adversarial exploitation, a common pitfall of overly complex models.

In summary, we believe using coresets is like cooking with the finest ingredients. It not only makes our ML model more efficient but also helps it build robustness, equipping it to tackle the ever-evolving landscape of cybersecurity threats.

7.3.3.5 The Power of Coresets in Action

Let us continue with our cooking example. Imagine trying to find the best ingredients for a recipe. We would not want to use everything in the pantry; instead, we would select the freshest and most essential items. Similarly, in ML, using a subset of the best data can make all the difference. This is where coresets come in. Coresets are like a highlight reel of our data. They focus on the most important and representative samples, making the training process both faster and more effective. This is particularly exciting for malware detection, where ensuring robustness against adversarial attacks is crucial. But how do we make this work?

Gradient Approximation: Imagine gradients as the lines that define shapes in an image, outlining the boundaries of objects. In malware detection, these "lines" help the model tell the difference between harmful and harmless behavior. Malware often has sharp lines, like distinct edges, because of unique features like suspicious API calls [189]. On the other hand, benign (harmless) samples are more like gentle slopes. If our coreset misses the sharp lines, it might overlook subtle malicious patterns. Conversely, if it ignores the smooth slopes of

[1] We thank Dr. Baharan Mirzasoleiman (UCLA) for granting permission to reproduce Figure 7.1, originally published in [244].

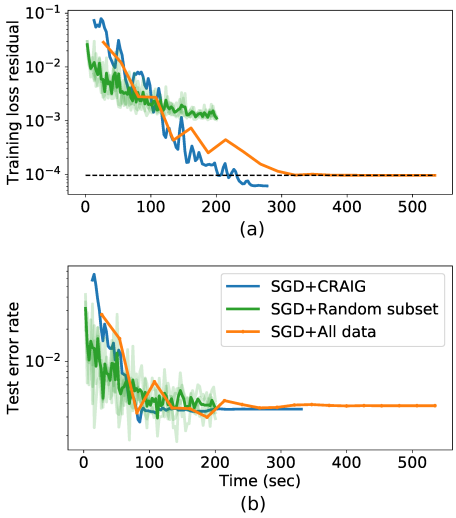


Figure 7.1: This graph illustrates the performance of an ML model in terms of error of Stochastic Gradient Descent for Logistic Regression over time, using different training methods on a dataset with approximately 580,000 data points [244]. The blue curve represents the model trained using CRAIG, a coreset construction technique that uses only 10% of the dataset but selects the most important data points. The green curve shows the model trained on a random 10% of the data, while the orange curve depicts the model trained on the entire dataset. CRAIG achieves similar performance to the full dataset, with the added benefit of being three times faster.

benign samples, it could result in false alarms. To be effective, coresets must capture the gradient behavior inherent to both malicious and benign data.

Designing Versatile Coresets: Just as a diverse cast of characters makes a story more engaging, diversity in key samples is vital for defining a coreset that enhances a model's robustness. In malware detection, it is crucial to reflect the variety and complexity of real-world samples, especially those tricky edge cases that hackers exploit. Therefore, coresets must cover critical regions of the data distribution that are susceptible to evasion attacks. By focusing on diversity and informativeness, models trained on these coresets can better generalize across both regular and tricky, adversarial examples. This approach aligns with findings from recent studies, which emphasize the importance of including diverse adversarial examples in training to bolster model robustness [246].

7.3.3.6 Last Word

Despite significant efforts to strengthen ML-based malware detection systems, improving their robustness remains a complex puzzle. While recent research has largely focused on enhancing algorithms, the quality of the data itself remains an often-overlooked factor. Here we highlight the need to shift our focus toward high-quality, representative data as the key to building robust malware classifiers. Think of data as the backbone of ML. High-quality data is essential for learning robust features, which, in turn, improve model robustness. But what exactly does this data include? It consists of the feature representations of real programs in the feature space. This means we also need to consider how feature representations shape

the distribution of data–ultimately influencing the decision boundary where malware is distinguished from benign software.

To move forward, our community must take a closer look at data quality, not just in terms of real programs but also in how they are represented in the feature space. By doing so, we can pave the way for more resilient ML systems that stand strong against evolving cyber threats.

Bibliography

References

[1] Ömer Aslan and Refik Samet. A comprehensive review on malware detection approaches. *IEEE Access*, 8:6249–6271, 2020.

[2] Daniel Gibert, Carles Mateu, and Jordi Planes. The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. *Journal of Network and Computer Applications*, 153:102526, 2020.

[3] Google Developers. Cloud-based protections. `https://developers.google.com/android/play-protect/cloud-based-protections`. Accessed: 2025-01-22.

[4] Pulei Xiong, Scott Buffett, Shahrear Iqbal, Philippe Lamontagne, Mohammad Mamun, and Heather Molyneaux. Towards a robust and trustworthy machine learning system development: An engineering perspective. *Journal of Information Security and Applications*, 65:103121, 2022.

[5] Zeinab Khorshidpour, Jafar Tahmoresnezhad, Sattar Hashemi, and Ali Hamzeh. Using domain adaptation in adversarial environment. *International Journal of Data Mining, Modelling and Management*, 9(3):201–219, 2017.

[6] Ling Huang, Anthony D Joseph, Blaine Nelson, Benjamin IP Rubinstein, and J Doug Tygar. Adversarial machine learning. In *Proceedings of the 4th ACM workshop on Security and artificial intelligence*, pages 43–58, 2011.

[7] Deqiang Li, Qianmu Li, Yanfang Ye, and Shouhuai Xu. Arms race in adversarial malware detection: A survey. *ACM Computing Surveys (CSUR)*, 55(1):1–35, 2021.

[8] Davide Maiorca, Ambra Demontis, Battista Biggio, Fabio Roli, and Giorgio Giacinto. Adversarial detection of flash malware: Limitations and open issues. *Computers & Security*, 96:101901, 2020.

[9] 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)*, pages 1332–1349. IEEE, 2020.

[10] Kevin Eykholt, Taesung Lee, Douglas Schales, Jiyong Jang, and Ian Molloy. {URET}: Universal Robustness Evaluation Toolkit (for Evasion). In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3817–3833, 2023.

[11] Heng Li, Zhang Cheng, Bang Wu, Liheng Yuan, Cuiying Gao, Wei Yuan, and Xiapu Luo. Black-box adversarial example attack towards FCG based Android malware detection under incomplete feature information. In *32nd USENIX Security Symposium (USENIX Security)*, 2023.

[12] Francesco Croce, Maksym Andriushchenko, Naman D Singh, Nicolas Flammarion, and Matthias Hein. Sparse-RS: a versatile framework for query-efficient sparse black-box adversarial attacks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 6437–6445, 2022.

[13] Jin Zhang, Chennan Zhang, Xiangyu Liu, Yuncheng Wang, Wenrui Diao, and Shanqing Guo. ShadowDroid: Practical Black-box Attack against ML-based Android Malware Detection. In *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 629–636. IEEE, 2021.

[14] Hemant Rathore, Sanjay K Sahay, Piyush Nikam, and Mohit Sewak. Robust Android malware detection system against adversarial attacks using Q-learning. *Information Systems Frontiers*, 23(4):867–882, 2021.

[15] Lingwei Chen, Shifu Hou, and Yanfang Ye. SecureDroid: Enhancing Security of Machine Learning-based Detection against Adversarial Android Malware Attacks. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 362–372, 2017.

[16] Ambra Demontis, Marco Melis, Battista Biggio, Davide Maiorca, Daniel Arp, Konrad Rieck, Igino Corona, Giorgio Giacinto, and Fabio Roli. Yes, machine learning can be more secure! a case study on Android malware detection. *IEEE Transactions on Dependable and Secure Computing*, 16(4):711–724, 2017.

[17] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial examples for malware detection. In *European symposium on research in computer security*, pages 62–79. Springer, 2017.

[18] Lingwei Chen, Shifu Hou, Yanfang Ye, and Shouhuai Xu. Droideye: Fortifying security of learning-based classifier against adversarial Android malware attacks. In *2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 782–789. IEEE, 2018.

[19] Xiaolei Liu, Xiaojiang Du, Xiaosong Zhang, Qingxin Zhu, Hao Wang, and Mohsen Guizani. Adversarial samples on Android malware detection systems for IoT systems. *Sensors*, 19(4):974, 2019.

[20] Guangquan Xu, GuoHua Xin, Litao Jiao, Jian Liu, Shaoying Liu, Meiqi Feng, and Xi Zheng. OFEI: A Semi-black-box Android Adversarial Sample Attack Framework Against DLaaS. *arXiv preprint arXiv:2105.11593*, 2021.

[21] Harel Berger, Chen Hajaj, and Amit Dvir. When the Guard failed the Droid: A case study of Android malware. *arXiv preprint arXiv:2003.14123*, 2020.

[22] Xiao Chen, Chaoran Li, Derui Wang, Sheng Wen, Jun Zhang, Surya Nepal, Yang Xiang, and Kui Ren. Android HIV: A study of repackaging malware for evading machine-learning detection. *IEEE Transactions on Information Forensics and Security*, 15:987–1001, 2019.

[23] Fabrizio Cara, Michele Scalas, Giorgio Giacinto, and Davide Maiorca. On the Feasibility of Adversarial Sample Creation Using the Android System API. *Information*, 11(9):433, 2020.

[24] Deqiang Li, Qianmu Li, Yanfang Ye, and Shouhuai Xu. A framework for enhancing deep neural networks against adversarial malware. *IEEE Transactions on Network Science and Engineering*, 8(1):736–750, 2021.

[25] Deqiang Li and Qianmu Li. Adversarial deep ensemble: Evasion attacks and defenses for malware detection. *IEEE Transactions on Information Forensics and Security*, 15:3886–3900, 2020.

[26] Guangquan Xu, Hongfei Shao, Jingyi Cui, Hongpeng Bai, Jiliang Li, Guangdong Bai, Shaoying Liu, Weizhi Meng, and Xi Zheng. GenDroid: A Query-Efficient Black-box Android Adversarial Attack Framework. *Computers & Security*, page 103359, 2023.

[27] Ping He, Yifan Xia, Xuhong Zhang, and Shouling Ji. Efficient query-based attack against ML-based Android malware detection under zero knowledge setting. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 90–104, 2023.

[28] Wei Yang, Deguang Kong, Tao Xie, and Carl A Gunter. Malware detection in adversarial settings: Exploiting feature evolutions and confusions in Android apps. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 288–302, 2017.

[29] Abdullah Al-Dujaili, Alex Huang, Erik Hemberg, and Una-May O'Reilly. Adversarial deep learning for robust detection of binary encoded malware. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 76–82. IEEE, 2018.

[30] Yi-Ming Chen, Chun-Hsien Yang, and Guo-Chung Chen. Using generative adversarial networks for data augmentation in Android malware detection. In *2021 IEEE conference on dependable and secure computing (DSC)*, pages 1–8. IEEE, 2021.

[31] Alex Huang, Abdullah Al-Dujaili, Erik Hemberg, and Una-May O'Reilly. On visual hallmarks of robustness to adversarial malware. *arXiv preprint arXiv:1805.03553*, 2018.

[32] Yonghong Huang, Utkarsh Verma, Celeste Fralick, Gabriel Infantec-Lopez, Brajesh Kumar, and Carl Woodward. Malware evasion attack and defense. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 34–38. IEEE, 2019.

[33] Aminollah Khormali, Ahmed Abusnaina, Songqing Chen, DaeHun Nyang, and Aziz Mohaisen. COPYCAT: practical adversarial attacks on visualization-based malware detection. *arXiv preprint arXiv:1909.09735*, 2019.

[34] Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. Adversarial malware binaries: Evading deep learning

for malware detection in executables. In *2018 26th European signal processing conference (EUSIPCO)*, pages 533–537. IEEE, 2018.

[35] Deqiang Li, Shicheng Cui, Yun Li, Jia Xu, Fu Xiao, and Shouhuai Xu. PAD: Towards principled adversarial malware detection against evasion attacks. *IEEE Transactions on Dependable and Secure Computing*, 21(2):920–936, 2023.

[36] Keane Lucas, Samruddhi Pai, Weiran Lin, Lujo Bauer, Michael K Reiter, and Mahmood Sharif. Adversarial training for {Raw-Binary} malware classifiers. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1163–1180, 2023.

[37] Chenyue Wang, Linlin Zhang, Kai Zhao, Xuhui Ding, and Xusheng Wang. AdvAndMal: Adversarial Training for Android Malware Detection and Family Classification. *Symmetry*, 13(6):1081, 2021.

[38] Guangquan Xu, GuoHua Xin, Litao Jiao, Jian Liu, Shaoying Liu, Meiqi Feng, and Xi Zheng. OFEI: A Semi-Black-Box Android Adversarial Sample Attack Framework Against DLaaS. *IEEE Transactions on Computers*, 2023.

[39] Battista Biggio, Giorgio Fumera, and Fabio Roli. Security evaluation of pattern classifiers under attack. *IEEE transactions on knowledge and data engineering*, 26(4):984–996, 2013.

[40] Alex Serban, Erik Poll, and Joost Visser. Adversarial examples on object recognition: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 53(3):1–38, 2020.

[41] Zeinab El-Rewini, Zhuo Zhang, and Yousra Aafer. Poirot: Probabilistically recommending protections for the android framework. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 937–950, 2022.

[42] Statista. Market share of smartphone operating systems worldwide from 2012 to 2024. `https://www.statista.com/statistics/272698/global-market-share-of-mobile-operating-systems/`, 2024.

[43] Juan Francisco Bertona. Beyond the App Store: The Hidden Risks of Sideloading Apps, 2024. Accessed: 2025-01-22. URL: `https://www.zimperium.com/blog/the-hidden-risks-of-sideloading-apps/`.

[44] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting Millions of Android Apps for the Research Community. In *ACM Mining Software Repositories (MSR)*, 2016.

[45] Ishai Rosenberg, Asaf Shabtai, Yuval Elovici, and Lior Rokach. Query-efficient black-box attack against sequence-based malware classifiers. In *Annual Computer Security Applications Conference*, pages 611–626, 2020.

[46] Luca Demetrio, Battista Biggio, Giovanni Lagorio, Fabio Roli, and Alessandro Armando. Functionality-preserving black-box optimization of adversarial Windows malware. *IEEE Transactions on Information Forensics and Security*, 16:3469–3478, 2021.

[47] Mahmood Sharif, Keane Lucas, Lujo Bauer, Michael K Reiter, and Saurabh Shintre. Optimization-guided binary diversification to mislead neural networks for malware detection. *arXiv preprint arXiv:1912.09064*, 2019.

[48] E Quiring, F Pendlebury, A Warnecke, F Pierazzi, C Wressnegger, L Cavallaro, and K Rieck. Dos and don'ts of machine learning in computer security. In *31st USENIX Security Symposium (USENIX Security 22), USENIX Association, Boston, MA*, 2022.

[49] Íñigo Íncer Romeo, 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*, pages 54–63, 2018.

[50] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. 2015.

[51] Roland S Zimmermann, Wieland Brendel, Florian Tramer, and Nicholas Carlini. Increasing confidence in adversarial robustness evaluations. *Advances in Neural Information Processing Systems*, 35:13174–13189, 2022.

[52] Shiqi Wang, Yizheng Chen, Ahmed Abdou, and Suman Jana. Mixtrain: Scalable training of verifiably robust neural networks. *arXiv preprint arXiv:1811.02625*, 2018.

[53] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *6th International Conference on Learning Representations, ICLR 2018- Conference Track Proceedings*, 2018.

[54] Xiaohan Zhang, Yuan Zhang, Ming Zhong, Daizong Ding, Yinzhi Cao, Yukun Zhang, Mi Zhang, and Min Yang. Enhancing State-of-the-art Classifiers with API Semantics to Detect Evolved Android Malware. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, pages 757–770, 2020.

[55] Paul C van Oorschot. Computer Security and the Internet, 2020.

[56] Kegan Mooney. Malware Statistics 2024: A Comprehensive Overview, May 2024. Updated on November 28, 2024, 5:26 AM. Accessed: 2024-11-28. URL: `https://www.next7it.com/insights/malware-statistics-2024-a-comprehensive-overview/`.

[57] Deqiang Li, Qianmu Li, Yanfang Ye, and Shouhuai Xu. Sok: Arms race in adversarial malware detection. *ACM Computing Surveys*, 55(1):15–35, 2021.

[58] Ashwini Mujumdar, Gayatri Masiwal, and Dr B Meshram. Analysis of signature-based and behavior-based anti-malware approaches. *International Journal of Advanced Research in Computer Engineering and Technology (IJARCET)*, 2(6):2037–2039, 2013.

[59] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles K Nicholas. Malware detection by eating a whole exe. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[60] Hyrum S Anderson, Anant Kharkar, Bobby Filar, David Evans, and Phil Roth. Learning to evade static pe machine learning malware models via reinforcement learning. *arXiv preprint arXiv:1801.08917*, 2018.

[61] Om Prakash Samantray and Satya Narayan Tripathy. IoT-malware classification model using byte sequences and supervised learning techniques. In *Next Generation of Internet of Things: Proceedings of ICNGIoT 2021*, pages 51–60. Springer, 2021.

[62] Microsoft. What is edr? endpoint detection and response, 2025. Accessed: 2025-01-24. URL: `https://www.microsoft.com/en-us/security/business/security-101/what-is-edr-endpoint-detection-response`.

[63] Tom Mitchell. Machine Learning. *Publisher: McGraw Hill*, 1997.

[64] Xingzhi Qian, Xinran Zheng, Yiling He, Shuo Yang, and Lorenzo Cavallaro. LAMD: Context-driven android malware detection and classification with LLMs. *arXiv preprint arXiv:2502.13055*, 2025.

[65] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):474–499, 2024.

[66] Masoud Mehrabi Koushki, Ibrahim AbuAlhaol, Anandharaju Durai Raju, Yang Zhou, Ronnie Salvador Giagone, and Huang Shengqiang. On building machine learning pipelines for Android malware detection: a procedural survey of practices, challenges and opportunities. *Cybersecurity*, 5(1):16, 2022.

[67] 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)*, pages 321–338, 2019.

[68] Olakunle Ibitoye, Rana Abou-Khamis, Mohamed el Shehaby, Ashraf Matrawy, and M Omair Shafiq. The Threat of Adversarial Attacks on Machine Learning in Network Security–A Survey. *arXiv preprint arXiv:1911.02621*, 2023.

[69] Battista Biggio, Blaine Nelson, and Pavel Laskov. Poisoning attacks against support vector machines. In *Proceedings of the 29th International Conference on Machine Learning (ICML)*, pages 1467–1474. Omnipress, 2012.

[70] Sen Chen, Minhui Xue, Lingling Fan, Lei Ma, Yang Liu, and Lihua Xu. How can we craft large-scale Android malware? An automated poisoning attack. In *2019 IEEE 1st international workshop on artificial intelligence for mobile (AI4Mobile)*, pages 21–24. IEEE, 2019.

[71] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[72] Octavian Suciu, Radu Marginean, Yigitcan Kaya, Hal Daume III, and Tudor Dumitras. When does machine learning {FAIL}? generalized transferability for evasion and poisoning attacks. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1299–1316, 2018.

[73] Robert Flood, Marco Casadio, David Aspinall, and Ekaterina Komendantskaya. Generating Traffic-Level Adversarial Examples from Feature-Level Specifications. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS) Workshops*, 2024. URL: `https://sites.google.com/view/secai2024/programme`.

[74] Ludwig Schmidt, Shibani Santurkar, Dimitris Tsipras, Kunal Talwar, and Aleksander Madry. Adversarially robust generalization requires more data. *Advances in neural information processing systems*, 31, 2018.

[75] Justin Gilmer, Nicolas Ford, Nicholas Carlini, and Ekin Cubuk. Adversarial examples are a natural consequence of test error in noise. In *International Conference on Machine Learning*, pages 2280–2289. PMLR, 2019.

[76] Viet Quoc Vo, Ehsan Abbasnejad, and Damith C Ranasinghe. RamBoAttack: A Robust Query Efficient Deep Neural Network Decision Exploit. In *Proceedings of the 29th Annual Network and Distributed System Security Symposium (NDSS)*, 2022. URL: `https://www.ndss-symposium.org/ndss-paper/auto-draft-239/`.

[77] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical Black-Box Attacks against Machine Learning. *Proceedings of the 12th ACM on Asia Conference on Computer and Communications Security*, pages 506–519, 2017.

[78] Kais Zaman, Mark McDonald, Sankaran Mahadevan, and Lawrence Green. Robustness-based design optimization under data uncertainty. *Structural and Multidisciplinary Optimization*, 44:183–197, 2011.

[79] Ruoxi Sun, Minhui Xue, Gareth Tyson, Tian Dong, Shaofeng Li, Shuo Wang, Haojin Zhu, Seyit Camtepe, and Surya Nepal. Mate! Are you really aware? An explainability-guided testing framework for robustness of malware detectors. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1573–1585, 2023.

[80] Yijun Yang, Ruiyuan Gao, Yu Li, Qiuxia Lai, and Qiang Xu. What You See is Not What the Network Infers: Detecting Adversarial Examples Based on Semantic Contradiction. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2022.

[81] Yuxin Wen, Shuai Li, and Kui Jia. Towards Understanding the Regularization of Adversarial Robustness on Neural Networks. In *Proceedings of the 37th International Conference on Machine Learning (ICML)*. PMLR, 2020.

[82] Weilin Xu, David Evans, and Yanjun Qi. Feature Squeezing: Detecting Adversarial Examples in Deep Neural Networks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2018.

[83] Mingfei Lu and Badong Chen. On the Adversarial Robustness of Generative Autoencoders in the Latent Space. *Neural Computing and Applications*, 36:8109–8123, 2024.

[84] Hamid Bostani and Veelasha Moonsamy. Evadedroid: A practical evasion attack on machine learning for black-box Android malware detection. *Computers & Security*, 139:1–18, 2024.

[85] Faraz Ahmed, Haider Hameed, M Zubair Shafiq, and Muddassar Farooq. Using spatio-temporal information in API calls with machine learning algorithms for malware detection. In *Proceedings of the 2nd ACM Workshop on Security and Artificial Intelligence*, pages 55–62, 2009.

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

[87] Mojtaba Eskandari, Zeinab Khorshidpour, and Sattar Hashemi. HDM-Analyser: a hybrid analysis approach based on data mining techniques for malware detection. *Journal of Computer Virology and Hacking Techniques*, 9(2):77–93, 2013.

[88] Jinrong Bai, Junfeng Wang, and Guozhong Zou. A malware detection scheme based on mining format information. *The Scientific World Journal*, 2014, 2014.

[89] Edward Raff and Charles Nicholas. An alternative to NCD for large sequences, Lempel-Ziv Jaccard distance. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1007–1015, 2017.

[90] Sitalakshmi Venkatraman, Mamoun Alazab, and R Vinayakumar. A hybrid deep learning image-based analysis for effective malware detection. *Journal of Information Security and Applications*, 47:377–389, 2019.

[91] Faranak Abri, Sima Siami-Namini, Mahdi Adl Khanghah, Fahimeh Mirza Soltani, and Akbar Siami Namin. Can machine/deep learning classifiers detect zero-day malware with high accuracy? In *2019 IEEE international conference on big data (Big Data)*, pages 3252–3259. IEEE, 2019.

[92] C. Castillo and R. Samani. McAfee Mobile Threat Report," *McAfee Advanced Threat Research and Mobile Malware Research Team, McAfee*. Technical report, McAfee, 2021.

[93] Deqiang Li, Qianmu Li, Yanfang Ye, and Shouhuai Xu. Enhancing deep neural networks against adversarial malware examples. *arXiv preprint arXiv:2004.07919*, 2020.

[94] Zhuo Ma, Haoran Ge, Yang Liu, Meng Zhao, and Jianfeng Ma. A combination method for Android malware detection based on control flow graphs and machine learning algorithms. *IEEE access*, 7:21235–21245, 2019.

[95] Lucky Onwuzurike, Enrico Mariconti, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version). *ACM Transactions on Privacy and Security (TOPS)*, 22(2):1–34, 2019.

[96] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. DREBIN: Effective and explainable detection of android malware in your pocket. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS 2014)*, volume 14, pages 1–15, 2014.

[97] Quentin Jerome, Kevin Allix, Radu State, and Thomas Engel. Using opcode-sequences to detect malicious Android applications. In *2014 IEEE international conference on communications (ICC)*, pages 914–919. IEEE, 2014.

[98] Wei Song, Xuezixiang Li, Sadia Afroz, Deepali Garg, Dmitry Kuznetsov, and Heng Yin. Automatic generation of adversarial examples for interpreting malware classifiers. pages 990–1003, 2022.

[99] Luca Demetrio, Battista Biggio, Giovanni Lagorio, Fabio Roli, and Alessandro Armando. Explaining vulnerabilities of deep learning to adversarial malware binaries. *arXiv preprint arXiv:1901.03583*, 2019.

[100] Apktool: a tool for reverse engineering Android apk files. `https://ibotpeaches.github.io/Apktool/`, 2010. Accessed: 2022-01-27.

[101] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, (CASCON 1999), pages 1–11. IBM Press, 1999.

[102] 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. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[103] Rosmalissa Jusoh, Ahmad Firdaus, Shahid Anwar, Mohd Zamri Osman, Mohd Faaizie Darmawan, and Mohd Faizal Ab Razak. Malware detection using static analysis in Android: a review of FeCO (features, classification, and obfuscation). *PeerJ Computer Science*, 7:e522, 2021.

[104] BooJoong Kang, Suleiman Y Yerima, Sakir Sezer, and Kieran McLaughlin. N-gram opcode analysis for Android malware detection. *arXiv preprint arXiv:1612.01445*, 2016.

[105] Kyoung Soo Han, Jae Hyun Lim, Boojoong Kang, and Eul Gyu Im. Malware analysis using visualized images and entropy graphs. *International Journal of Information Security*, 14:1–14, 2015.

[106] Earl T Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 257–269, 2015.

[107] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 421–430. IEEE, 2007.

[108] Nicholas Carlini, Anish Athalye, Nicolas Papernot, Wieland Brendel, Jonas Rauber, Dimitris Tsipras, Ian Goodfellow, Aleksander Madry, and Alexey Kurakin. On evaluating adversarial robustness. *arXiv preprint arXiv:1902.06705*, 2019.

[109] Robert Moskovitch, Dima Stopel, Clint Feher, Nir Nissim, and Yuval Elovici. Unknown malcode detection via text categorization and the imbalance problem. In *2008 IEEE international conference on intelligence and security informatics*, pages 156–161. IEEE, 2008.

[110] Sachin Jain and Yogesh Kumar Meena. Byte level n–gram analysis for malware detection. In *International Conference on Information Processing*, pages 51–59. Springer, 2011.

[111] Asaf Shabtai, Robert Moskovitch, Clint Feher, Shlomi Dolev, and Yuval Elovici. Detecting unknown malicious code by applying classification techniques on opcode patterns. *Security Informatics*, 1(1):1–22, 2012.

[112] Igor Santos, Felix Brezo, Xabier Ugarte-Pedrero, and Pablo G Bringas. Opcode sequences as representation of executables for data-mining-based unknown malware detection. *Information Sciences*, 231:64–82, 2013.

[113] Zhang Fuyong and Zhao Tiezhu. Malware detection and classification based on n-grams attribute similarity. In *2017 IEEE international conference on computational science and engineering (CSE) and IEEE international conference on embedded and ubiquitous computing (EUC)*, volume 1, pages 793–796. IEEE, 2017.

[114] Gerardo Canfora, Andrea De Lorenzo, Eric Medvet, Francesco Mercaldo, and Corrado Aaron Visaggio. Effectiveness of opcode ngrams for detection of multi family Android malware. In *2015 10th International Conference on Availability, Reliability and Security*, pages 333–340. IEEE, 2015.

[115] MV Varsha, P Vinod, and KA Dhanya. Identification of malicious Android app using manifest and opcode features. *Journal of Computer Virology and Hacking Techniques*, 13(2):125–138, 2017.

[116] MZ Mas'ud, S Sahib, MF Abdollah, SR Selamat, and R Yusof. An evaluation of N-gram system call sequence in mobile malware detection. *ARPN J. Eng. Appl. Sci*, 11(5):3122–3126, 2016.

[117] Takia Islam, Sheikh Shah Mohammad Motiur Rahman, Md Aumit Hasan, Abu Sayed Md Mostafizur Rahaman, and Md Ismail Jabiullah. Evaluation of N-gram based multi-layer approach to detect malware in Android. *Procedia Computer Science*, 171:1074–1082, 2020.

[118] LA Rastrigin. The convergence of the random search method in the extremal control of a many parameter system. *Automaton & Remote Control*, 24:1337–1342, 1963.

[119] Mark Weiser. Program slicing. *IEEE Transactions on software engineering*, (4):352–357, 1984.

[120] Luis Muñoz-González and Emil C Lupu. The security of machine learning systems. In *AI in Cybersecurity*, pages 47–79. Springer, 2019.

[121] Jeonguk Ko, Hyungjoon Shim, Dongjin Kim, Youn-Sik Jeong, Seong-je Cho, Minkyu Park, Sangchul Han, and Seong Baeg Kim. Measuring similarity of Android applications via reversing and K-gram birthmarking. In *Proceedings of the 2013 Research in Adaptive and Convergent Systems*, pages 336–341. 2013.

[122] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 659–674, 2015.

[123] Yousra Aafer, Wenliang Du, and Heng Yin. DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. In *International conference on security and privacy in communication systems*, pages 86–103. Springer, 2013.

[124] Ashkan Sami, Babak Yadegari, Hossein Rahimi, Naser Peiravian, Sattar Hashemi, and Ali Hamze. Malware detection based on mining API calls. In *Proceedings of the 2010 ACM symposium on applied computing*, pages 1020–1025, 2010.

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

[126] VirusTotal. VirusTotal. `https://www.virustotal.com`, 2004. Accessed: 2022-09-11.

[127] Aleieldin Salem. Towards accurate labeling of android apps for reliable malware detection. In *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy*, pages 269–280, 2021.

[128] 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)*, pages 729–746, 2019.

[129] AV-TEST Institute. AV-TEST. https://www.av-test.org/en/antivirus/mobile-devices, 2004. Accessed: 2022-09-11.

[130] Fabrício Ceschin, Marcus Botacin, Heitor Murilo Gomes, Luiz S Oliveira, and André Grégio. Shallow security: On the creation of adversarial variants to evade machine learning-based malware detectors. In *Proceedings of the 3rd Reversing and Offensive-oriented Trends Symposium*, pages 1–9, 2019.

[131] UI/Application Exerciser Monkey Kernel Description. https://developer.android.com/studio/test/other-testing-tools/app-crawler?authuser=1, 2023. Accessed: 2023-06-06.

[132] Michael Spreitzenbarth. DREBIN Feature Extractor. https://www.dropbox.com/s/ztthwf6ub4mxxc9/feature-extractor.tar.gz, 2014. Accessed: 2022-09-11.

[133] Intriguing Properties of Adversarial ML Problem-Space Attacks. https://s2lab.cs.ucl.ac.uk/projects/intriguing, 2020. Accessed: 2024-08-17.

[134] MaMaDroid Source Code. https://bitbucket.org/gianluca_students/mamadroid_code, 2020. Accessed: 2022-09-11.

[135] Adversarial Deep Ensemble for Malware Detection. https://github.com/deqangss/adv-dnn-ens-malware, 2020. Accessed: 2022-09-11.

[136] Moustafa Alzantot, Yash Sharma, Supriyo Chakraborty, Huan Zhang, Cho-Jui Hsieh, and Mani B Srivastava. GenAttack: Practical black-box attacks with gradient-free optimization. In *Proceedings of the genetic and evolutionary computation conference*, pages 1111–1119, 2019.

[137] Hamid Bostani, Zhengyu Zhao, Zhuoran Liu, and Veelasha Moonsamy. Level Up with ML Vulnerability Identification: Leveraging Domain Constraints in Feature Space for Robust Android Malware Detection. *ACM Transactions on Privacy and Security*, 28(2):1–32, 2025.

[138] Hui-Juan Zhu, Zhu-Hong You, Ze-Xuan Zhu, Wei-Lei Shi, Xing Chen, and Li Cheng. DroidDet: effective and robust detection of Android malware using static analysis along with rotation forest model. *Neurocomputing*, 272:638–646, 2018.

[139] Zhenlong Yuan, Yongqiang Lu, and Yibo Xue. DroidDetector: Android malware characterization and detection using deep learning. *Tsinghua Science and Technology*, 21(1):114–123, 2016.

[140] Win Zaw Zarni Aung. Permission-based Android malware detection. *International Journal of Scientific & Technology Research*, 2(3):228–234, 2013.

[141] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of Android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 45–54, 2013.

[142] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. DroidMat: Android Malware Detection through Manifest and API Calls Tracing. In *2012 Seventh Asia Joint Conference on Information Security*, pages 62–69. IEEE, 2012.

[143] Bo Li and Yevgeniy Vorobeychik. Evasion-robust classification on binary domains. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 12(4):1–32, 2018.

[144] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2014.

[145] Fabio Carrara, Rudy Becarelli, Roberto Caldelli, Fabrizio Falchi, and Giuseppe Amato. Adversarial examples detection in features distance spaces. In *Proceedings of the European conference on computer vision (ECCV) workshops*, pages 1–15, 2018.

[146] Shawn Shan. Using honeypots to catch adversarial attacks on neural networks. In *Proceedings of the 8th ACM Workshop on Moving Target Defense*, pages 1–25, 2021.

[147] Kathrin Grosse, Praveen Manoharan, Nicolas Papernot, Michael Backes, and Patrick McDaniel. On the (statistical) detection of adversarial examples. *arXiv preprint arXiv:1702.06280*, 2017.

[148] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *2018 International Conference on Learning Representations (ICLR)*, 2018.

[149] 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)*, pages 582–597. IEEE, 2016.

[150] Salijona Dyrmishi, Salah Ghamizi, Thibault Simonetto, Yves Le Traon, and Maxime Cordy. On The Empirical Effectiveness of Unrealistic Adversarial Hardening Against Realistic Adversarial Attacks. In *2023 IEEE symposium on security and privacy (SP)*, 2023.

[151] Thibault Simonetto, Salijona Dyrmishi, Salah Ghamizi, Maxime Cordy, and Yves Le Traon. A Unified Framework for Adversarial Attack and Defense in Constrained Feature Space. In *Thirty-First International Joint Conference on Artificial Intelligence IJCAI-22*, pages 1313–1319, 2022.

[152] Ryan Sheatsley, Nicolas Papernot, Michael Weisman, Gunjan Verma, and Patrick McDaniel. Adversarial examples in constrained domains. *arXiv preprint arXiv:2011.01183*, 2020.

[153] Bao Gia Doan, Shuiqiao Yang, Paul Montague, Olivier De Vel, Tamas Abraham, Seyit Camtepe, Salil S Kanhere, Ehsan Abbasnejad, and Damith C Ranasinghe. Feature-Space Bayesian Adversarial Learning Improved Malware Detector Robustness. In *AAAI*, 2023.

[154] Raphael Labaca-Castro, Luis Muñoz-González, Feargus Pendlebury, Gabi Dreo Rodosek, Fabio Pierazzi, and Lorenzo Cavallaro. Realizable Universal Adversarial Perturbations for Malware. *arXiv preprint arXiv:2102.06747*, 2021.

[155] Weiwei Hu and Ying Tan. Black-box attacks against RNN based malware detection algorithms. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[156] Keane Lucas, Mahmood Sharif, Lujo Bauer, Michael K Reiter, and Saurabh Shintre. Malware makeover: Breaking ml-based static analysis by modifying executable bytes. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, pages 744–758, 2021.

[157] 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 *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 490–510. Springer, 2018.

[158] Joao P Papa, Alexandre X Falcao, and Celso TN Suzuki. Supervised pattern classification based on optimum-path forest. *International Journal of Imaging Systems and Technology*, 19(2):120–131, 2009.

[159] Heng Li, Shiyao Zhou, Wei Yuan, Xiapu Luo, Cuiying Gao, and Shuiyan Chen. Robust Android malware detection against adversarial example attacks. In *Proceedings of the Web Conference 2021*, pages 3603–3612, 2021.

[160] 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. *Computers & Security*, 86:168–182, 2019.

[161] Ecenaz Erdemir, Jeffrey Bickford, Luca Melis, and Sergul Aydore. Adversarial robustness with non-uniform perturbations. *Advances in Neural Information Processing Systems (NeurIPS)*, 34:19147–19159, 2021.

[162] Jiyu Chen, David Wang, and Hao Chen. Explore the transformation space for adversarial images. In *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*, pages 109–120, 2020.

[163] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 ieee symposium on security and privacy (sp)*, pages 39–57. IEEE, 2017.

[164] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *2016 IEEE European symposium on security and privacy (EuroS&P)*, pages 372–387. IEEE, 2016.

[165] Giovanni Apruzzese, Mauro Conti, and Ying Yuan. Spacephish: The evasion-space of adversarial attacks against phishing website detectors using machine learning. In *Proceedings of the 38th Annual Computer Security Applications Conference*, pages 171–185, 2022.

[166] Giovanni Apruzzese, Mauro Andreolini, Luca Ferretti, Mirco Marchetti, and Michele Colajanni. Modeling realistic adversarial attacks against network intrusion detection systems. *Digital Threats: Research and Practice (DTRAP)*, 3(3):1–19, 2022.

[167] Ryan Sheatsley, Blaine Hoak, Eric Pauley, Yohan Beugin, Michael J Weisman, and Patrick McDaniel. On the robustness of domain constraints. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 495–515, 2021.

[168] Martin Teuffenbach, Ewa Piatkowska, and Paul Smith. Subverting Network Intrusion Detection: Crafting Adversarial Examples Accounting for Domain-Specific Constraints. In *International Cross-Domain Conference for Machine Learning and Knowledge Extraction*, pages 301–320. Springer, 2020.

[169] Alesia Chernikova and Alina Oprea. Fence: Feasible evasion attacks on neural networks in constrained environments. *arXiv preprint arXiv:1909.10480*, 2019.

[170] Liang Tong, Bo Li, Chen Hajaj, Chaowei Xiao, Ning Zhang, and Yevgeniy Vorobeychik. Improving robustness of ML classifiers against realizable evasion attacks using conserved features. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 285–302, 2019.

[171] Shifu Hou, Yanfang Ye, Yangqiu Song, and Melih Abdulhayoglu. Hindroid: An intelligent Android malware detection system based on structured heterogeneous information network. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1507–1515, 2017.

[172] Ke Xu, Yingjiu Li, Robert H Deng, and Kai Chen. Deeprefiner: Multi-layer android malware detection system applying deep neural networks. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 473–487. IEEE, 2018.

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

[174] Yunzhe Tian, Yingdi Wang, Endong Tong, Wenjia Niu, Liang Chang, Qi Alfred Chen, Gang Li, and Jiqiang Liu. Exploring Data Correlation between Feature Pairs for Generating Constraint-based Adversarial Examples. In *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 430–437. IEEE, 2020.

[175] Harel Berger, Chen Hajaj, and Amit Dvir. Evasion is not enough: A case study of Android malware. In *International Symposium on Cyber Security Cryptography and Machine Learning*, pages 167–174. Springer, 2020.

[176] Hamid Bostani, Mansour Sheikhan, and Behrad Mahboobi. A strong coreset algorithm to accelerate OPF as a graph-based machine learning in large-scale problems. *Information Sciences*, 555:424–441, 2021.

[177] Leonardo Marques Rocha, Fábio AM Cappabianco, and Alexandre Xavier Falcão. Data clustering as an optimum-path forest problem with applications in image analysis. *International Journal of Imaging Systems and Technology*, 19(2):50–68, 2009.

[178] Ian Goodfellow, Patrick McDaniel, and Nicolas Papernot. Making machine learning robust against adversarial inputs. *Communications of the ACM*, 61(7):56–66, 2018.

[179] Huimin Zeng, Chen Zhu, Tom Goldstein, and Furong Huang. Are adversarial examples created equal? a learnable weighted minimax risk for robustness under non-uniform attacks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 10815–10823, 2021.

[180] Chen Liu, Bo Li, Jun Zhao, Weiwei Feng, Xudong Liu, and Chunpei Li. A2-CLM: Few-Shot Malware Detection Based on Adversarial Heterogeneous Graph Augmentation. *IEEE Transactions on Information Forensics and Security*, 2023.

[181] Florian Tramer and Dan Boneh. Adversarial training and robustness for multiple perturbations. *Advances in neural information processing systems*, 32, 2019.

[182] Puyudi Yang, Jianbo Chen, Cho-Jui Hsieh, Jane-Ling Wang, and Michael I Jordan. Greedy attack and gumbel attack: Generating adversarial examples for discrete data. *Journal of Machine Learning Research*, 21(43):1–36, 2020.

[183] Fengjuan Gao, Yu Wang, and Ke Wang. Discrete adversarial attack to models of code. *Proceedings of the ACM on Programming Languages*, 7(PLDI):172–195, 2023.

[184] Nadia Daoudi, Kevin Allix, Tegawendé François Bissyandé, and Jacques Klein. A deep dive inside drebin: An explorative analysis beyond Android malware detection scores. *ACM Transactions on Privacy and Security*, 25(2):1–28, 2022.

[185] Lucky Onwuzurike, Mario Almeida, Enrico Mariconti, Jeremy Blackburn, Gianluca Stringhini, and Emiliano De Cristofaro. A family of Droids-Android malware detection via behavioral modeling: Static vs dynamic analysis. In *2018 16th Annual Conference on Privacy, Security and Trust (PST)*, pages 1–10. IEEE, 2018.

[186] David Freedman, Robert Pisani, and Roger Purves. Statistics (international student edition). *Pisani, R. Purves, 4th edn. WW Norton & Company, New York*, 2007.

[187] Nadia Daoudi, Jordan Samhi, Abdoul Kader Kabore, Kevin Allix, Tegawendé F Bissyandé, and Jacques Klein. DexRay: A Simple, yet Effective Deep Learning Approach to Android Malware Detection Based on Image Representation of Bytecode. In *Deployable Machine Learning for Security Defense: Second International Workshop,*

MLHat 2021, Virtual Event, August 15, 2021, Proceedings 2, pages 81–106. Springer, 2021.

[188] Geoffrey I Webb, Roy Hyde, Hong Cao, Hai Long Nguyen, and Francois Petitjean. Characterizing concept drift. *Data Mining and Knowledge Discovery*, 30(4):964–994, 2016.

[189] 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)*, pages 524–528. IEEE, 2018.

[190] Maura Pintor, Luca Demetrio, Angelo Sotgiu, Ambra Demontis, Nicholas Carlini, Battista Biggio, and Fabio Roli. Indicators of attack failure: Debugging and improving optimization of adversarial examples. *Advances in Neural Information Processing Systems*, 35:23063–23076, 2022.

[191] Hamid Bostani, Zhengyu Zhao, and Veelasha Moonsamy. Improving adversarial robustness in android malware detection by reducing the impact of spurious correlations. In *European Symposium on Research in Computer Security*, pages 204–222. Springer, 2024.

[192] Huan Zhang, Hongge Chen, Zhao Song, Duane Boning, Inderjit S Dhillon, and Cho-Jui Hsieh. The limitations of adversarial training and the blind-spot attack. In *International Conference on Learning Representations (ICLR)*, 2019.

[193] He He, Sheng Zha, and Haohan Wang. Unlearn Dataset Bias in Natural Language Inference by Fitting the Residual. In *Proceedings of the 2nd Workshop on Deep Learning Approaches for Low-Resource NLP (DeepLo 2019)*, pages 132–142, Hong Kong, China, November 2019. Association for Computational Linguistics. URL: https://aclanthology.org/D19-6115, doi:10.18653/v1/D19-6115.

[194] Wenqian Ye, Guangtao Zheng, Xu Cao, Yunsheng Ma, Xia Hu, and Aidong Zhang. Spurious Correlations in Machine Learning: A Survey. *arXiv preprint arXiv:2402.12715*, 2024.

[195] Corinna Cortes, Mehryar Mohri, Michael Riley, and Afshin Rostamizadeh. Sample selection bias correction theory. In *International conference on algorithmic learning theory*, pages 38–53. Springer, 2008.

[196] Shai Ben-David, John Blitzer, Koby Crammer, and Fernando Pereira. Analysis of representations for domain adaptation. *Advances in neural information processing systems*, 19, 2006.

[197] Weiwei Hu and Ying Tan. Generating adversarial malware examples for black-box attacks based on GAN. In *International Conference on Data Mining and Big Data*, pages 409–423. Springer, 2022.

[198] Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Logan Engstrom, Brandon Tran, and Aleksander Madry. Adversarial examples are not bugs, they are features. *Advances in neural information processing systems*, 32, 2019.

[199] Martin Arjovsky, Léon Bottou, Ishaan Gulrajani, and David Lopez-Paz. Invariant risk minimization. *arXiv preprint arXiv:1907.02893*, 2019.

[200] Wanqian Yang, Polina Kirichenko, Micah Goldblum, and Andrew G Wilson. Chroma-vae: Mitigating shortcut learning with generative classifiers. *Advances in Neural Information Processing Systems*, 35:20351–20365, 2022.

[201] Ruimeng Li, Yuanhao Pu, Zhaoyi Li, Hong Xie, and Defu Lian. Invariant Representation Learning via Decoupling Style and Spurious Features. *arXiv preprint arXiv:2312.06226*, 2023.

[202] Anders Sandberg, Stuart Armstrong, Rebecca Gorman, and Rei England. Sigmoids behaving badly: why they usually cannot predict the future as well as they seem to promise. *arXiv preprint arXiv:2109.08065*, 2021.

[203] Hamid Bostani and Mansour Sheikhan. Hybrid of binary gravitational search algorithm and mutual information for feature selection in intrusion detection systems. *Soft computing*, 21(9):2307–2324, 2017.

[204] Hamid Bostani, Jacopo Cortellazzi, Daniel Arp, Fabio Pierazzi, Veelasha Moonsamy, and Lorenzo Cavallaro. On the Effectiveness of Adversarial Training on Malware Classifiers. *arXiv preprint arXiv:2412.18218*, 2024.

[205] Qinglong Wang, Wenbo Guo, Kaixuan Zhang, Alexander G Ororbia, Xinyu Xing, Xue Liu, and C Lee Giles. Adversary resistant deep neural networks with an application to malware detection. In *Proceedings of the 23rd ACM sigkdd international conference on knowledge discovery and data mining*, pages 1145–1153, 2017.

[206] Salijona Dyrmishi, Salah Ghamizi, Thibault Simonetto, Yves Le Traon, and Maxime Cordy. On the empirical effectiveness of unrealistic adversarial hardening against realistic adversarial attacks. In *2023 IEEE symposium on security and privacy (SP)*, pages 1384–1400. IEEE, 2023.

[207] Yunchun Zhang, Haorui Li, Yang Zheng, Shaowen Yao, and Jiaqi Jiang. Enhanced DNNs for malware classification with GAN-based adversarial training. *Journal of Computer Virology and Hacking Techniques*, 17:153–163, 2021.

[208] Xiruo Wang and Risto Miikkulainen. MDEA: Malware detection with evolutionary adversarial learning. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE, 2020.

[209] Hamid Eghbal-zadeh, Werner Zellinger, Maura Pintor, Kathrin Grosse, Khaled Koutini, Bernhard A Moser, Battista Biggio, and Gerhard Widmer. Rethinking data augmentation for adversarial robustness. *Information Sciences*, 654:119838, 2024.

[210] Jacopo Cortellazzi, Feargus Pendlebury, Daniel Arp, Erwin Quiring, Fabio Pierazzi, and Lorenzo Cavallaro. Intriguing Properties of Adversarial ML Attacks in the Problem Space [Extended Version], 2024. URL: `https://arxiv.org/abs/1911.02142`, `arXiv:1911.02142`.

[211] Battista Biggio and Fabio Roli. Wild patterns: Ten years after the rise of adversarial machine learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2154–2156, 2018.

[212] 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*, pages 387–402. Springer, 2013.

[213] Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. Analysis of classifiers' robustness to adversarial perturbations. *Machine learning*, 107(3):481–508, 2018.

[214] Pedro Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87, 2012.

[215] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008.

[216] Xinran Zheng, Shuo Yang, Edith CH Ngai, Suman Jana, and Lorenzo Cavallaro. Learning temporal invariance in android malware detectors. *arXiv preprint arXiv:2502.05098*, 2025.

[217] Matan Levi and Aryeh Kontorovich. Splitting the Difference on Adversarial Training. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1163–1180, 2023.

[218] Hongyang Zhang, Yaodong Yu, Jiantao Jiao, Eric Xing, Laurent El Ghaoui, and Michael Jordan. Theoretically principled trade-off between robustness and accuracy. In *International Conference on Machine Learning*, pages 7472–7482. PMLR, 2019.

[219] Jiapeng Zhao, Zhongjin Liu, Xiaoling Zhang, Jintao Huang, Zhiqiang Shi, Shichao Lv, Hong Li, and Limin Sun. Gradient-based adversarial attacks against malware detection by instruction replacement. In *International Conference on Wireless Algorithms, Systems, and Applications*, pages 603–612. Springer, 2022.

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

[221] Ke Xu, Yingjiu Li, Robert Deng, Kai Chen, and Jiayun Xu. Droidevolver: Self-evolving android malware detection system. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 47–62. IEEE, 2019.

[222] 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*, Santa Clara, CA, 2019. USENIX Association. USENIX Sec.

[223] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *NDSS*, 2014.

[224] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic Differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[225] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-Learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[226] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A Library for Large Linear Classification. *J. Mach. Learn. Res.*, 9:1871–1874, 2008.

[227] Nicholas Frosst and Geoffrey Hinton. Distilling a neural network into a soft decision tree. *arXiv preprint arXiv:1711.09784*, 2017.

[228] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2623–2631, 2019.

[229] Hamid Bostani and Veelasha Moonsamy. Beyond Learning Algorithms: The Crucial Role of Data in Robust Malware Detection. *IEEE Security & Privacy*, 23:1–6, 2025.

[230] Farida Siddiqi Prity, Md Shahidul Islam, Emran Hossain Fahim, Md Maruf Hossain, Sazzad Hossain Bhuiyan, Md Ariful Islam, and Mirza Raquib. Machine learning-based cyber threat detection: an approach to malware detection and security with explainable AI insights. *Human-Intelligent Systems Integration*, pages 1–30, 2024.

[231] AV-TEST Institute. Malware statistics & trends report, 2024. Accessed: 2025-02-08. URL: `https://www.av-test.org/en/statistics/malware/`.

[232] Bo Li, Peng Qi, Bo Liu, Shuai Di, Jingen Liu, Jiquan Pei, Jinfeng Yi, and Bowen Zhou. Trustworthy AI: From principles to practices. *ACM Computing Surveys*, 55(9):1–46, 2023.

[233] Suleiman Y Yerima, Mohammed K Alzaylaee, and Sakir Sezer. Machine learning-based dynamic analysis of android apps with improved code coverage. *EURASIP Journal on Information Security*, 2019(1):1–24, 2019.

[234] Palo Alto Networks. Why you need static analysis, dynamic analysis, and machine learning. Accessed: 2025-02-03. URL: `https://www.paloaltonetworks.com/cyberpedia/why-you-need-static-analysis-dynamic-analysis-machine-learning`.

[235] M. Krol. GitHub besieged by millions of malicious repositories in ongoing attack. *Ars Technica*, 2024. Accessed: 2025-01-30. URL: https://arstechnica.com/security/2024/02/.

[236] Snyk Team. GitHub Malware Repositories and Repo Confusion. *Snyk*, 2024. Accessed: 2025-01-30. URL: https://snyk.io/blog/github-malware-repositories-repo-confusion/.

[237] GitHub Security Team. Leveraging Machine Learning to Find Security Vulnerabilities. *GitHub Blog*, 2024. Accessed: 2025-01-30. URL: https://github.blog/security/vulnerability-research/leveraging-machine-learning-find-security-vulnerabilities/.

[238] Daniel E Geer Jr. Data. *IEEE Security & Privacy*, 23:98–100, 2025.

[239] Marco Loog, Tom Viering, and Alexander Mey. Minimizers of the empirical risk and risk monotonicity. *Advances in Neural Information Processing Systems*, 32, 2019.

[240] Marco Loog, Jesse H Krijthe, and Manuele Bicego. Also for k-means: more data does not imply better performance. *Machine Learning*, 112(8):3033–3050, 2023.

[241] Shiori Sagawa, Aditi Raghunathan, Pang Wei Koh, and Percy Liang. An investigation of why overparameterization exacerbates spurious correlations. In *International Conference on Machine Learning*, pages 8346–8356. PMLR, 2020.

[242] Devansh Arpit, Stanislaw Jastrzebski, Nicolas Ballas, David Krueger, Emmanuel Bengio, Maxinder S. Kanwal, Tegan Maharaj, Asja Fischer, Aaron Courville, Yoshua Bengio, and Simon Lacoste-Julien. A Closer Look at Memorization in Deep Networks. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, pages 233–242. PMLR, 2017.

[243] Gereon Frahling and Christian Sohler. Coresets in dynamic geometric data streams. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 209–217, 2005.

[244] Baharan Mirzasoleiman, Jeff Bilmes, and Jure Leskovec. Coresets for data-efficient training of machine learning models. In *International Conference on Machine Learning*, pages 6950–6960. PMLR, 2020.

[245] Baharan Mirzasoleiman, Kaidi Cao, and Jure Leskovec. Coresets for robust training of deep neural networks against noisy labels. *Advances in Neural Information Processing Systems*, 33:11465–11477, 2020.

[246] Yunseok Jang, Tianchen Zhao, Seunghoon Hong, and Honglak Lee. Adversarial defense via learning to generate diverse attacks. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 2740–2749, 2019.

About the Author

Hamid Bostani was born on November 29, 1984, in Shiraz, Iran. He earned his Bachelor's and Master's degrees in Computer Engineering (Software Engineering) from the Shiraz and South Tehran branches of Islamic Azad University (IAU), Iran, in 2008 and 2015, respectively. His Master's thesis, titled "Intrusion Detection and Identification of Attacks on the Internet of Things Using a Combination of Machine Learning Methods", was supervised by Prof. Mansour Sheikhan and received the Best Master's Thesis Award at the 5th Research, Scientific & Technological National Festival of IAU in 2017. Additionally, in December 2017, he was honored with the Outstanding Researcher Award at the Annual Convention of Plaudits for Top-tier Researchers at the South Tehran Branch of IAU. Hamid joined the Digital Security group at the Institute for Computing and Information Sciences, Radboud University, The Netherlands, in October 2020, where he conducted research on the security of Machine Learning (ML) in malware detection under the supervision of Prof. Veelasha Moonsamy and Dr. Erik Poll. During his PhD, he was a visiting scholar in the Cybersecurity Group at King's College London and the Systems Security Lab at University College London in the UK from October 2023 to March 2024, where he studied the robustness of ML-based malware detection under the supervision of Dr. Fabio Pierazzi and Prof. Lorenzo Cavallaro. In addition to his academic background, Hamid has industrial experience as a Software Engineer at the National Organization for Educational Testing (NOET) in Iran from 2012 to 2020. In 2019, he received the Best Employee Award at NOET. Hamid is currently in the process of being hired as a Postdoctoral Researcher at the Interdisciplinary Centre for Security, Reliability, and Trust at the University of Luxembourg to continue working on ML security under the supervision of Dr. Maxime Cordy.

Radboud Universiteit